Dependable Computing in Deep Space Applications For Tokyo Institute of Technology

Savio Chau, Ph.D.
Supervisor/Principal Engineer
Advanced Concept and Architecture Group
Jet Propulsion Laboratory

Agenda

- Challenges for Dependable Computing in Deep Space Environment
- Dependable Computing in Previous Deep Space Missions
- Dependable Computing in New Generation of Deep Space Mission
- Current Dependable Computing Researches in Jet Propulsion Laboratory
  - Fault-Tolerant High Performance Bus Architecture
  - Guarded Software Upgrade
- Direction of Dependable Computing Researches for Future Flight Missions

Deep Space Challenges for Dependable Computing

- Environmental Challenges
  - Extreme temperatures
  - Very strong radiation in some environments (e.g., Jupiter)
  - Not possible to replace failed components
- Resource Limitation
  - Power, especially for missions beyond Mars
  - Mass is limited by launch cost and flight time
  - Cost: space qualified components are expensive and therefore it is strongly desirable to reduce redundancy
- Operation
  - Any failures can have serious consequence
  - Difficult to diagnose problems with remote commanding
  - Spacecraft travels at very high speed and commands must be executed in real time
  - Communication bandwidth is low (in the order of a few kbps) and latency is long (4 hours one-way delay time)
- It has to work! Failure is not acceptable both financially and politically

Dependable Computing in Previous Deep Space Missions

- Characteristics of Previous Deep Space Missions:
  - Most missions were fly-by or orbiter
  - Most of the missions requirements were not very demanding
    - Images were gathered at relatively low rate (a few images per min)
    - Most of the other science data were relatively low volume
    - Operations were relatively simple, so that command sequences could be pre-programmed on ground and then uploaded to spacecraft
    - Fault detection and recovery is more manageable (although not easy)
  - The spacecrafts were physically large
    - Able to carry more mass and power for redundancy
  - Large mission operation team
    - Monitor telemetry continuously
    - Able to respond to crisis rapidly

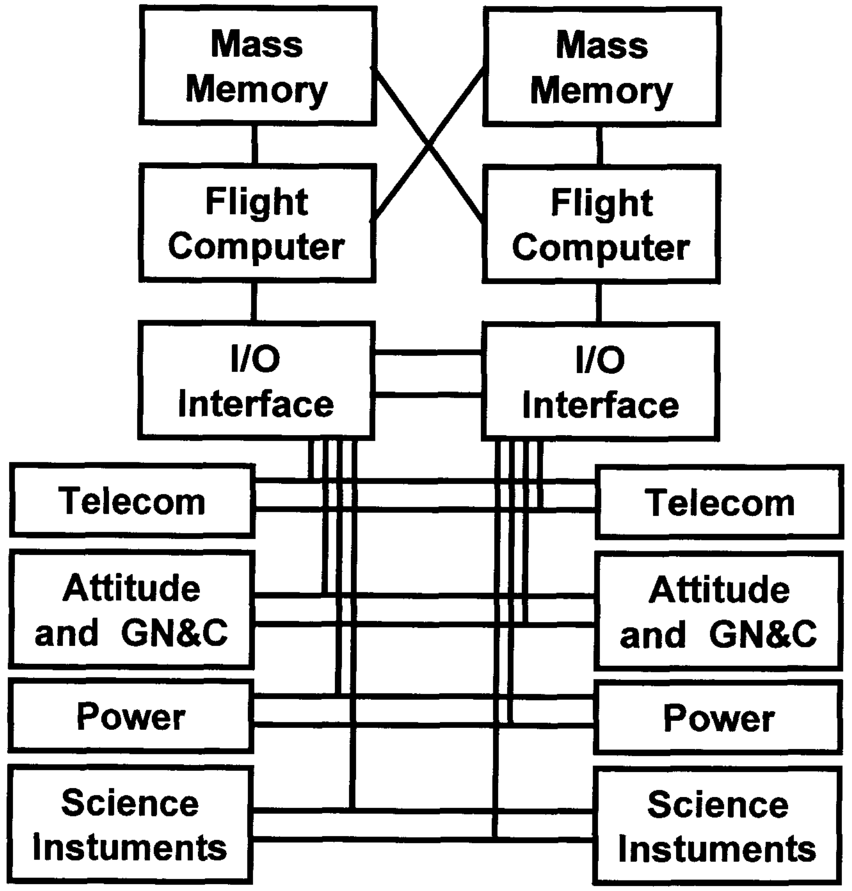
Dependable Computing in Previous Deep Space Missions

- Dependable Computing Techniques in Previous Deep Space Missions
  - Emphasized on fault protection rather than fault tolerance: Just to ensure faults would not cause the loss of the spacecraft. Uninterrupt operation was not required.
  - Faults usually were detected on-board, while ground operation was involved in fault isolation and recovery.
    - Safing: after a fault is detected, the spacecraft shuts down all but the most critical functions, search for a reference point (e.g., the Sun), point the antenna toward Earth, and wait for futhre commands from ground.
  - Error and fault containment regions
    - Ensure faults are detected locally
    - Ensure the failed fault containment region can be powered off without affect the rest of the system
  - Dual-string design: duplicate the entire set of subsystems (string) and cross-strap the components of both strings

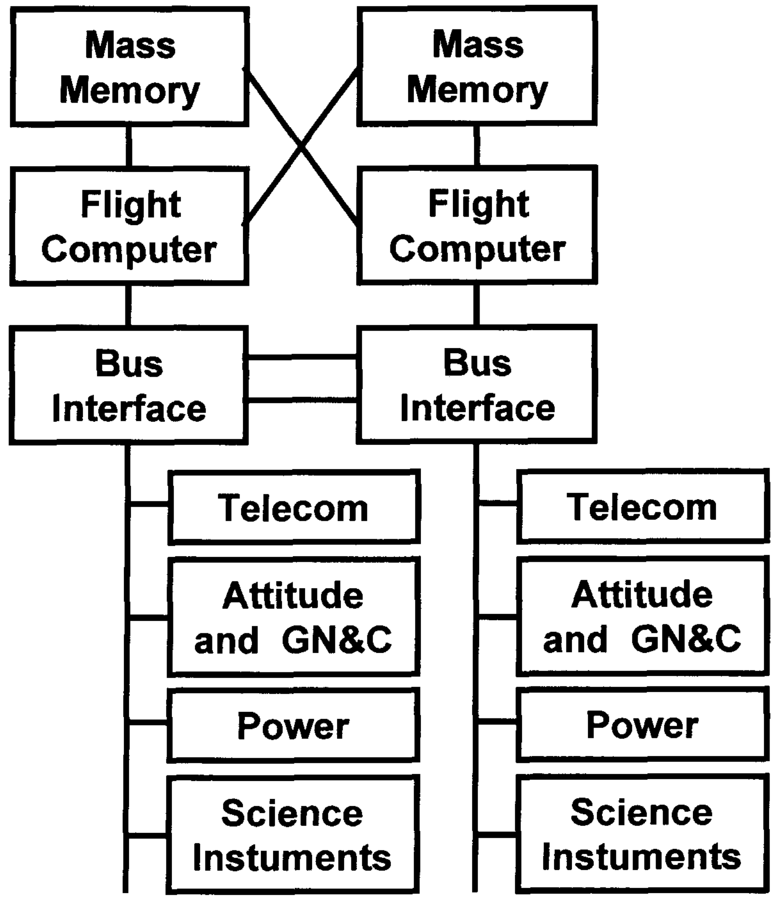
Dependable Computing in Previous Deep Space Missions

- Typical Dual-String Designs for Spacecraft

Point-to-Point Architecture



Bus-Based Architecture



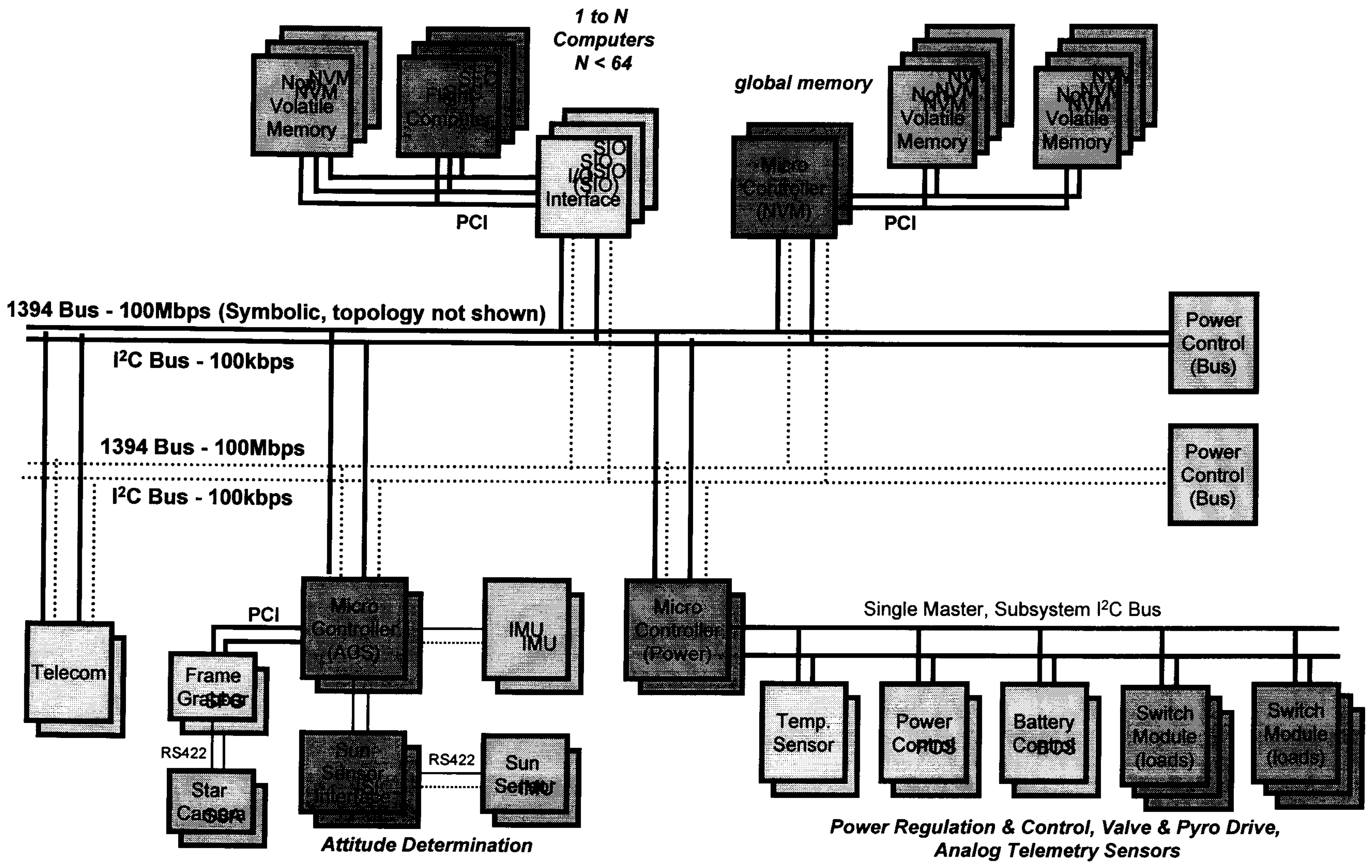
Dependable Computing in New Generation of Deep Space Mission

- Characteristics of New Generation of Deep Space Missions:
  - Many missions focus on landers, rovers, and sample return
  - Missions requirements are much more demanding
    - Precision landing
    - Hazard avoidance
    - Much higher processing requirements
    - Distributed processing
    - Much higher interface bandwidth requirements
    - Autonomous operation
    - High speed fault detection and recovery
  - The systems are physically smaller
  - Shrinking mission operation budget means smaller mission operation team
    - Must rely on on-board autonomous fault tolerance

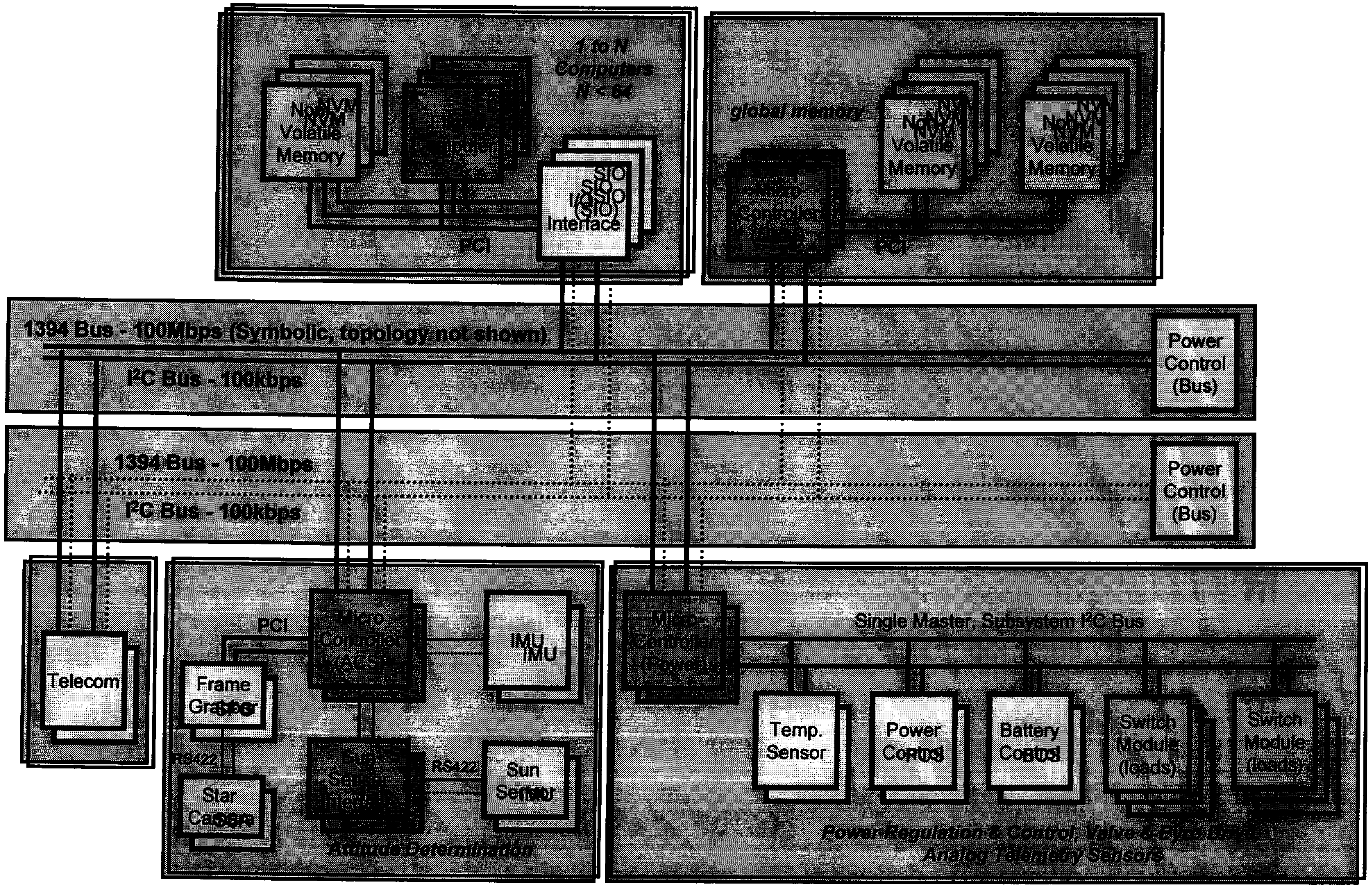
High-Performance Fault-Tolerant Bus Architecture Research at JPL

- Requirements for Distributed Processing, High Interface Bandwidth, and High Speed Fault Recovery Necessitate the Development of a High Speed Fault Tolerant Bus Architecture
- Commercial-Off-The-Shelf (COTS) Bus Standards Are Highly Desirable Because of Cost, Availability, and Performance Benefits.
- Two COTS Bus Standards Were Selected: the IEEE 1394 and I2C
- However, These COTS Buses Are Not Designed for the Highly Reliable Applications Such As Deep Space Missions. Therefore, <u>the Focus of the Research is How to Achieve Highly Reliable Avionics Bus Architecture Using COTS Bus Standards</u>
- The High Performance Fault-Tolerant Bus Architecture is Part of a NASA Funded Technology Program Called X2000

Baseline X2000 COTS Bus Architecture



X2000 Fault Containment Regions



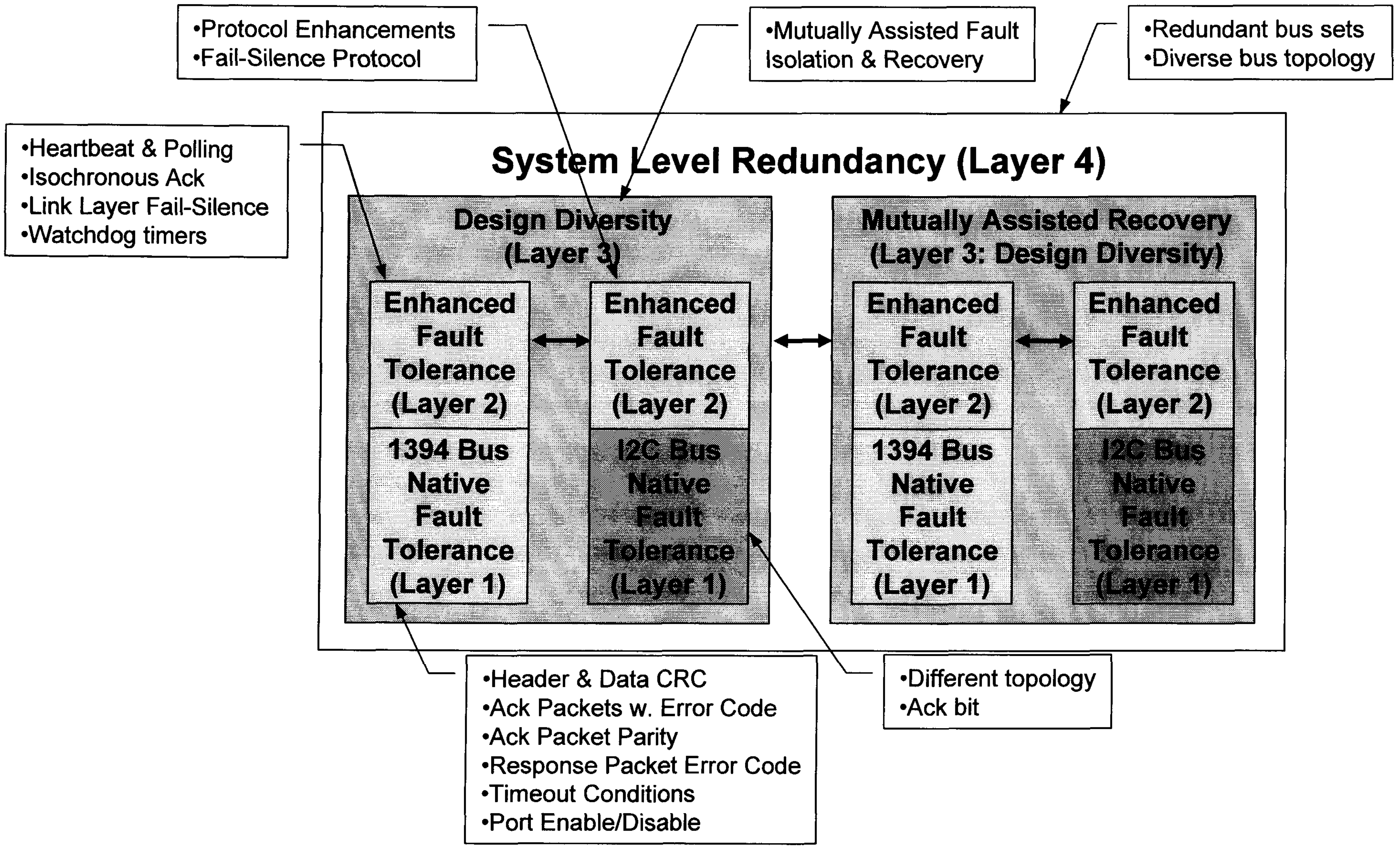
Benefits of Using COTS for Deep Space

- **Cost!**
  - **Lower development cost**
    - Commercial components or intellectual properties reduce design cost
    - Commercial software are available for widely supported COTS hardware
    - Architecture based on commercial standards can be used for multiple missions
  - **Reduced recurring cost**
    - Standard system components can be "mass produced" for multiple missions
  - **Lower integration and test costs**
    - Commercial designs are widely tested by the market
  - **Lower cost for test equipment**
    - No need to develop test equipment; they are commercially available
- **Availability**
  - Space qualified parts are difficult to find nowadays
- **Performance**
  - COTS performance usually is leading space qualified parts due to market pressure

Challenges of Using COTS for Deep Space

- Commercial components usually do not meet space environmental requirements
  - e.g., extreme temperatures, radiation, shock & vibration, long mission life
- Commercial design techniques might not be suitable for space applications
  - e.g., dynamic logic such as pre-charge circuits are not suitable for high radiation environments
- Many commercial designs do not have sufficient fault tolerance for effective fault containment and recovery
- Commercial vendors do not like to change their designs for a “narrow” market of reliable computing

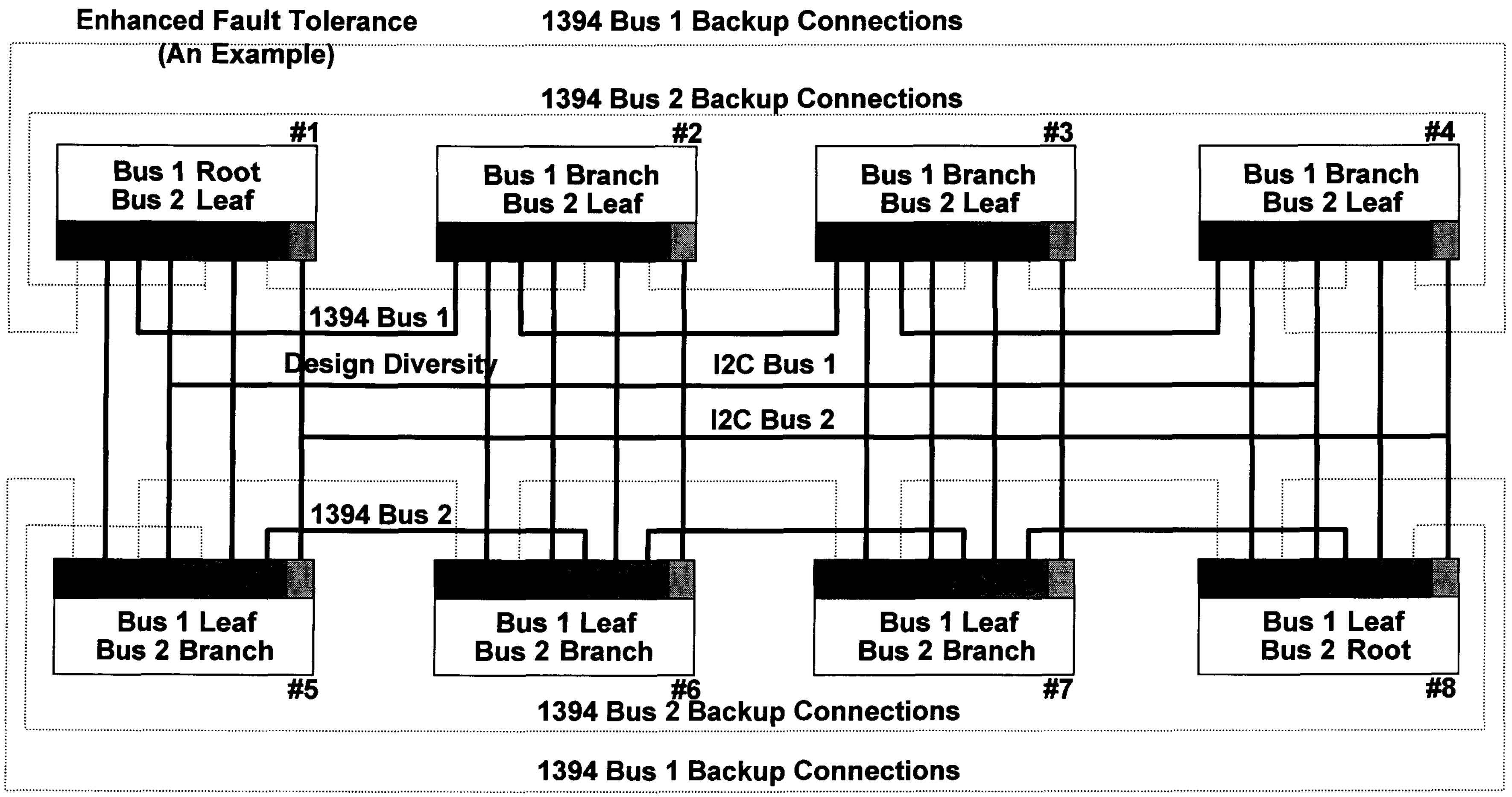
Multi-Layer Fault Tolerance Methodology for COTS-Based Bus Architecture



•Protocol Enhancements
•Fail-Silence Protocol
•Mutually Assisted Fault Isolation & Recovery
•Redundant bus sets
•Diverse bus topology
•Heartbeat & Polling
•Isochronous Ack
•Link Layer Fail-Silence
•Watchdog timers
System Level Redundancy (Layer 4)
Design Diversity (Layer 3)
Mutually Assisted Recovery (Layer 3: Design Diversity)
Enhanced Fault Tolerance (Layer 2)
Enhanced Fault Tolerance (Layer 2)
Enhanced Fault Tolerance (Layer 2)
Enhanced Fault Tolerance (Layer 2)
1394 Bus Native Fault Tolerance (Layer 1)
I2C Bus Native Fault Tolerance (Layer 1)
1394 Bus Native Fault Tolerance (Layer 1)
I2C Bus Native Fault Tolerance (Layer 1)
•Header & Data CRC
•Ack Packets w. Error Code
•Ack Packet Parity
•Response Packet Error Code
•Timeout Conditions
•Port Enable/Disable
•Different topology
•Ack bit

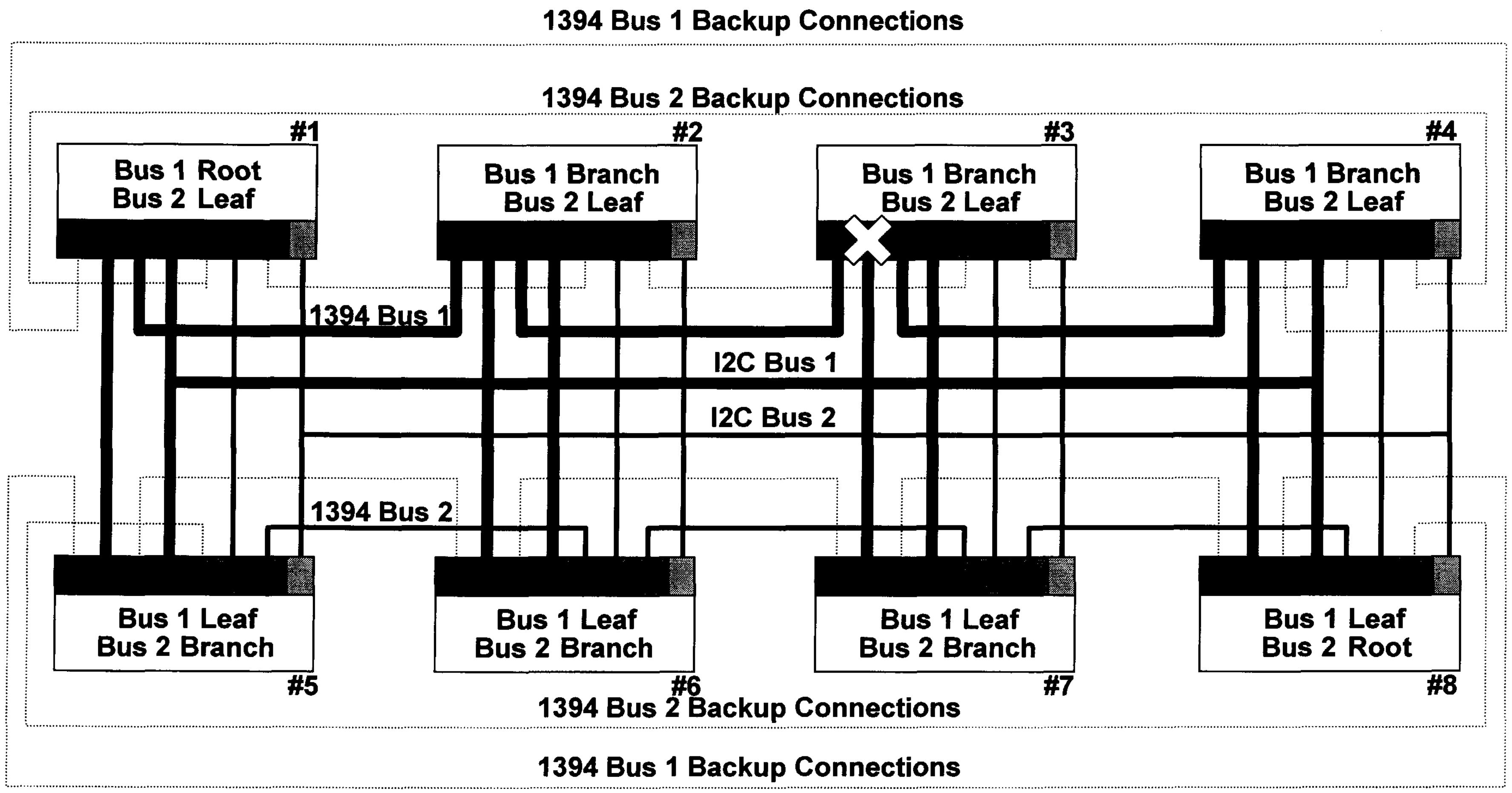
Realization of Multi-Level Fault Protection Methodology

Cable IEEE 1394 has a tree topology

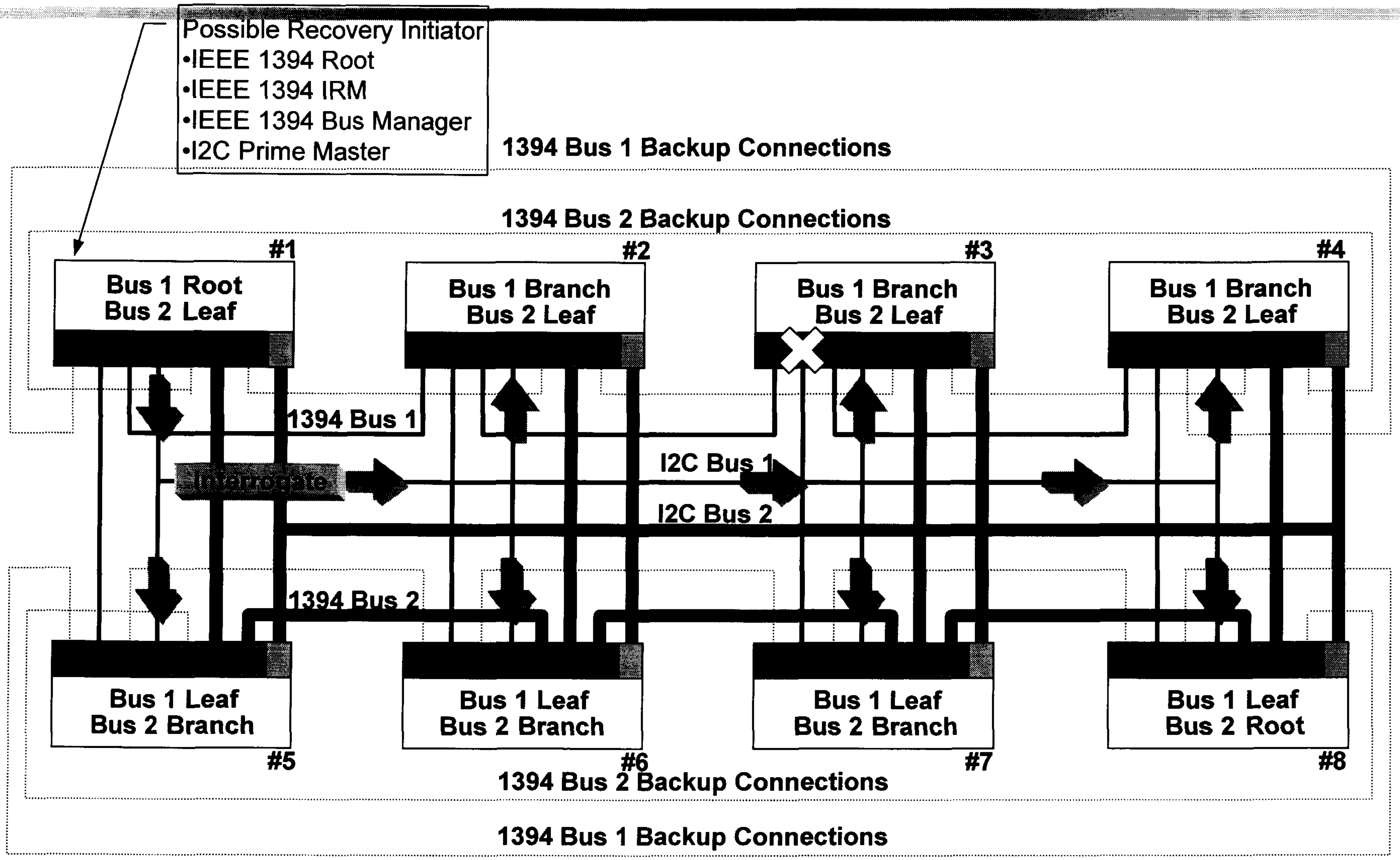


System Level Redundancy with Diverse Topology

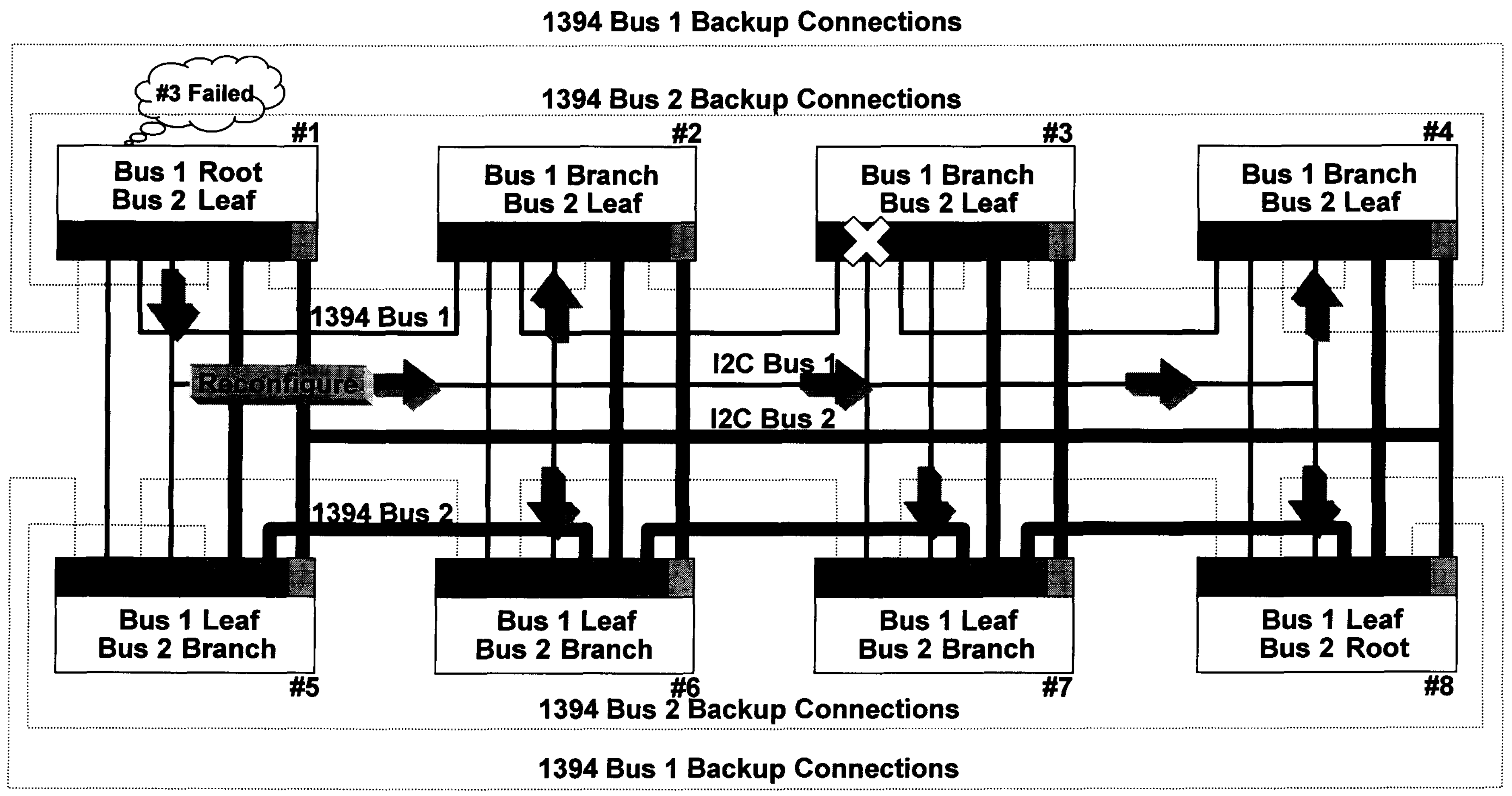
X2000 Fault Protection Strategy



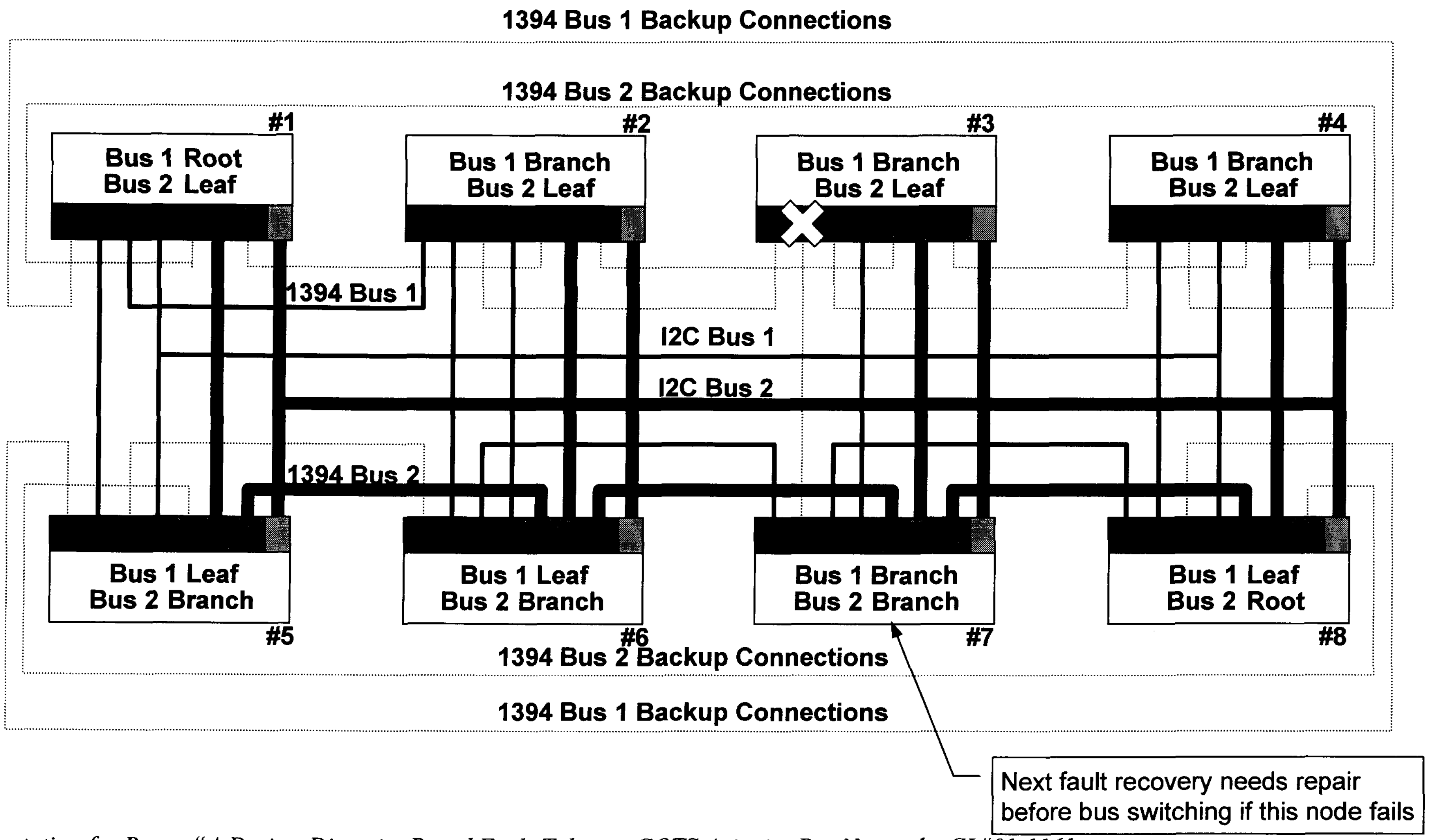
X2000 Fault Protection Strategy



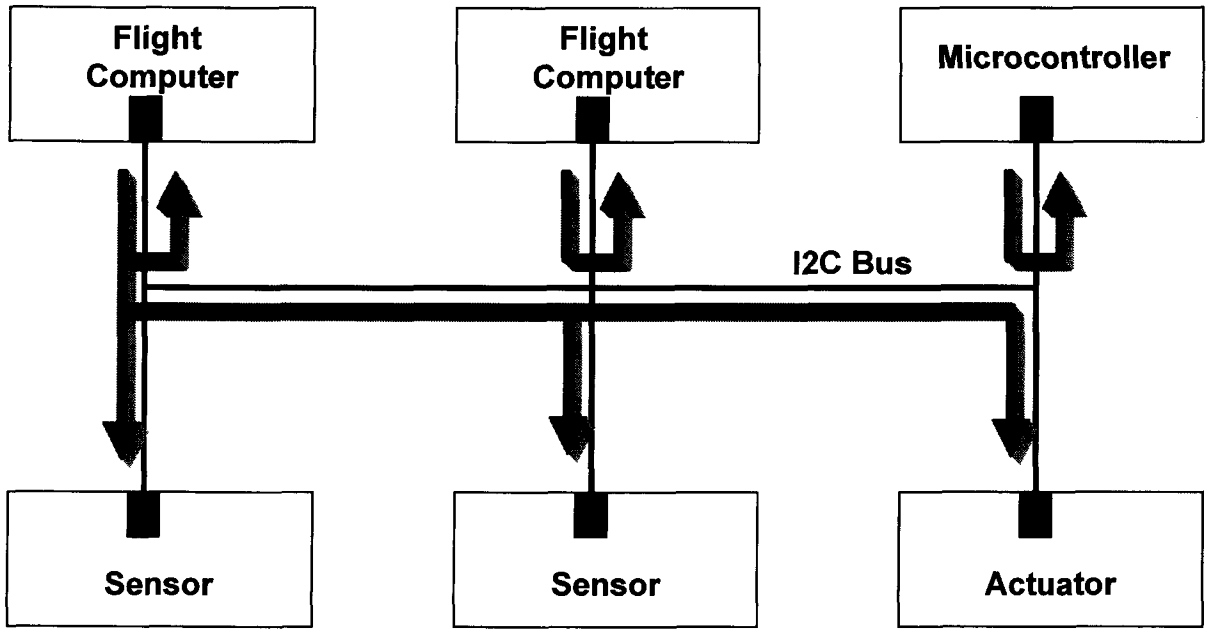
X2000 Fault Protection Strategy



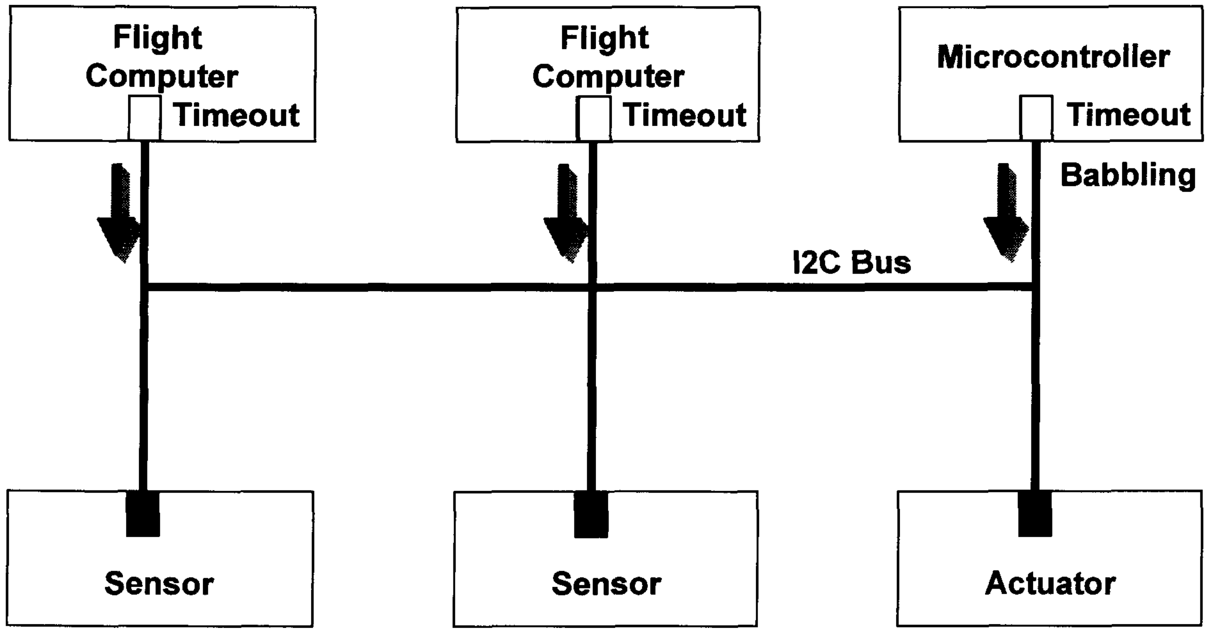
X2000 Fault Protection Strategy



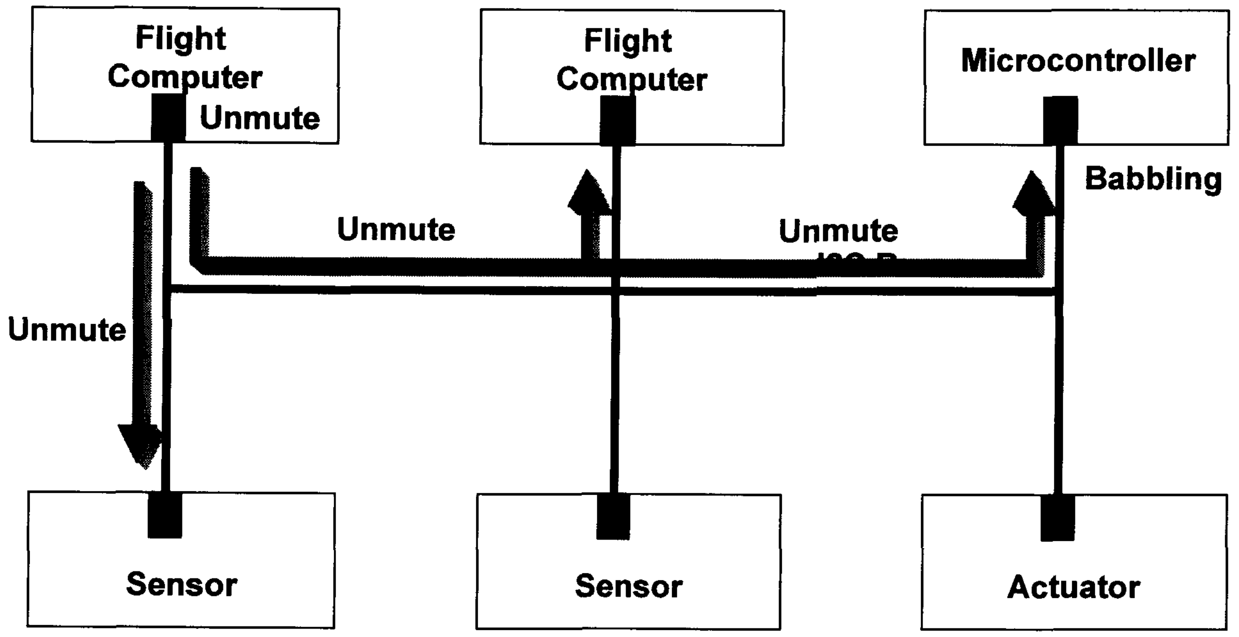
I^{2}C Bus Fault Protection: Fail Silence



I^2C Bus Fault Protection: Fail Silence



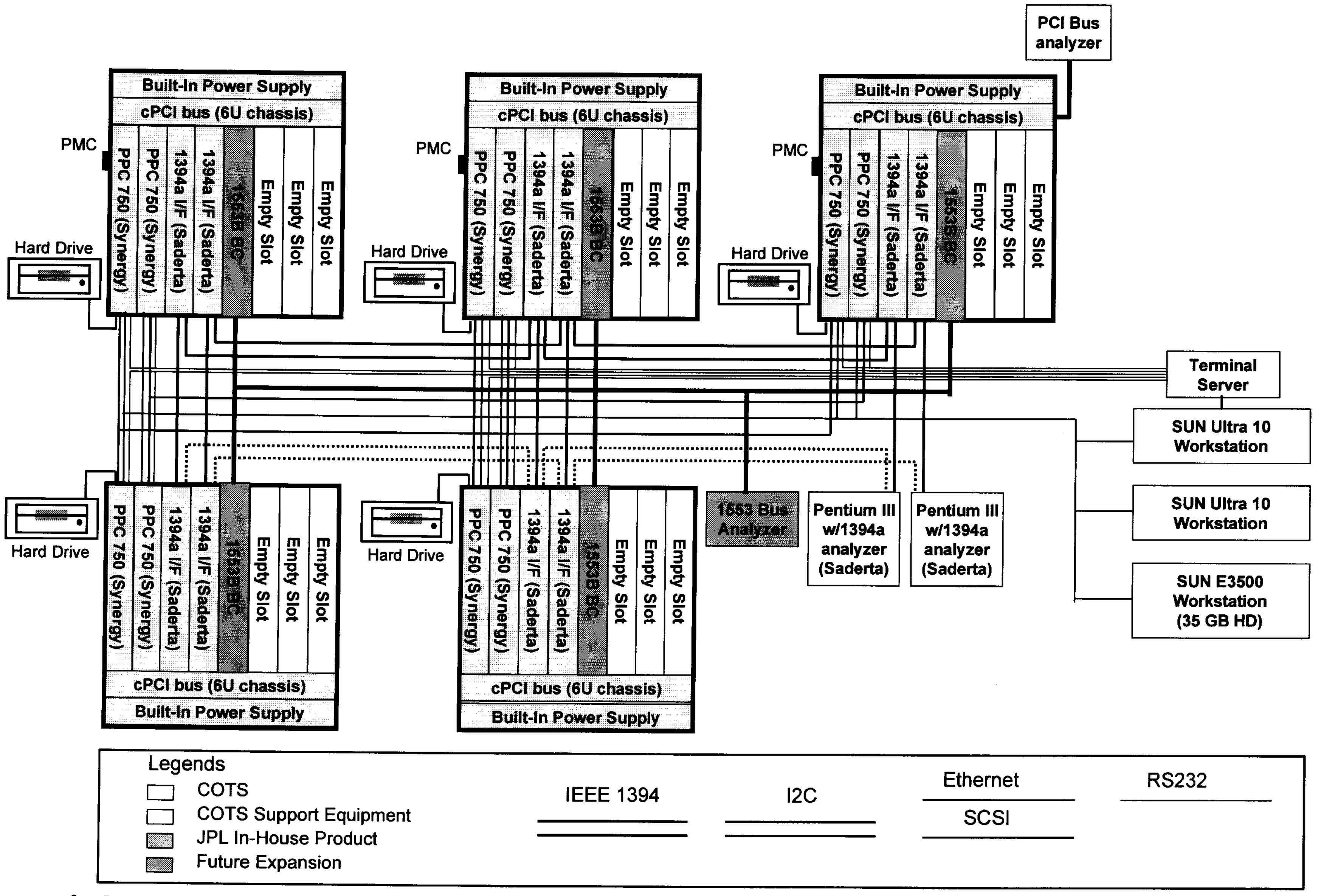
I^{2}C Bus Fault Protection: Fail Silence



Status of Implementation

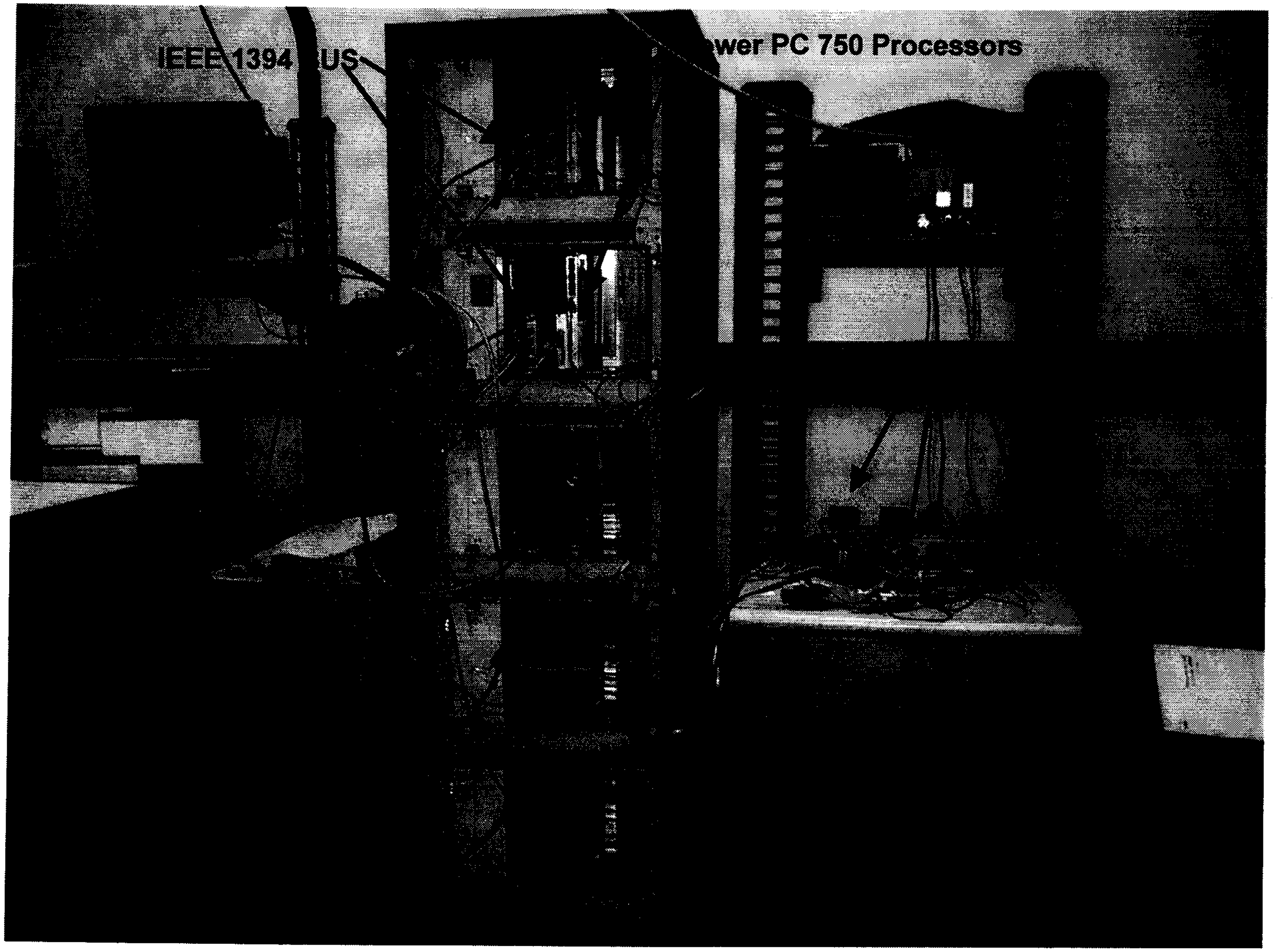
- Prototype (FPGA) of the IEEE 1394/I2C Bus Interface Board Has Been Designed and Fabricated. Testing is Underway.
- Rad-Hard (1 Mrad) ASICs for the IEEE 1394/I2C Link Layer and Physical Layer Will be Fabricated by Honeywell in the 4th Quarter of 2002
- A Flight Version Testbed Has been Set Up for Prototyping the Bus Architecture for Flight Projects
- A Commercial Version Testbed Has Been Set Up for IEEE 1394/I2C Architecture Experiments and Fault Injection

Architecture Testbed Configuration



Presentation for Paper "A Multi-Mission Testbed for Advanced Technologies". CL#01-0238

Architecture Testbed

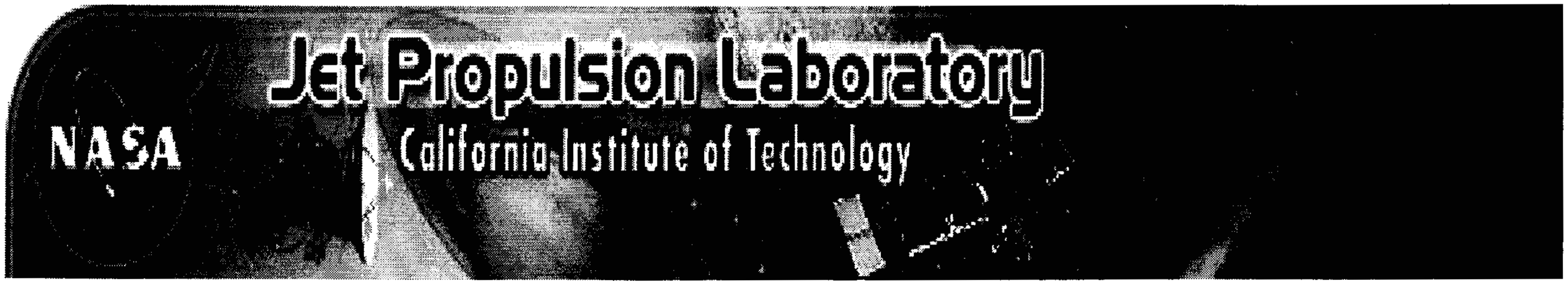


Fault Injection on the IEEE 1394 Bus

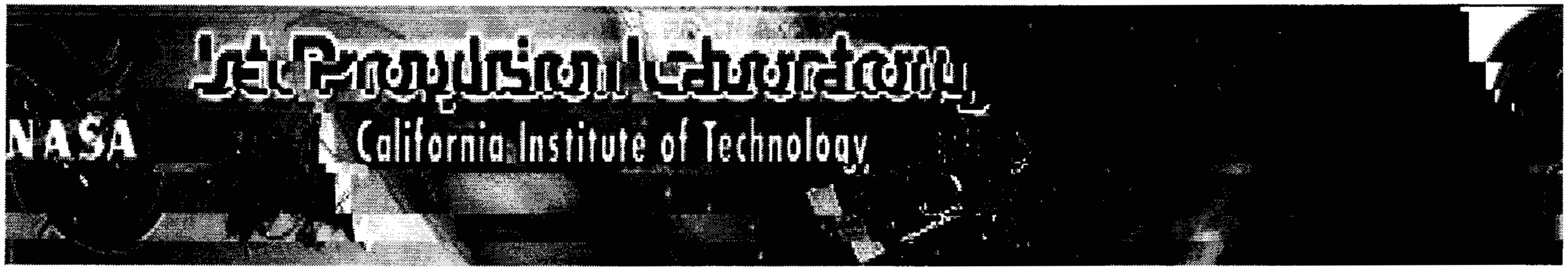
- Collaboration with Prof. Ravi Iyer in University of Illinois, Urbana-Champaign
- Purpose is to gain better understanding of the IEEE 1394 bus behavior under fault conditions. The knowledge will be used to improve the IEEE 1394 bus fault tolerance design
- Both hardware and software fault injection will be implemented:
  - Software fault injection injects faults into registers of the bus interface. It is done by the NFTAPE.
  - Hardware fault injection injects faults into the bus media. A custom board is designed for the fault injection.
- Status:
  - Software fault injection has been implemented in the CISM Future Deliveries Testbed
  - Hardware fault injector has been designed but not fabricated due to funding reduction
- Current Plan is to Continue the Software Fault Injection Campaign this Year and Postpone the Fabrication of the Hardware Fault Injector

Example of Fault Injection in IEEE 1394 Bus

Fault Injected in the Gap Count Register in the IEEE 1394 Bus

(A)


Before Fault Injection

(B)


After Fault Injection

Next Step in High-Performance Fault-Tolerant Avionics Bus Architecture Research

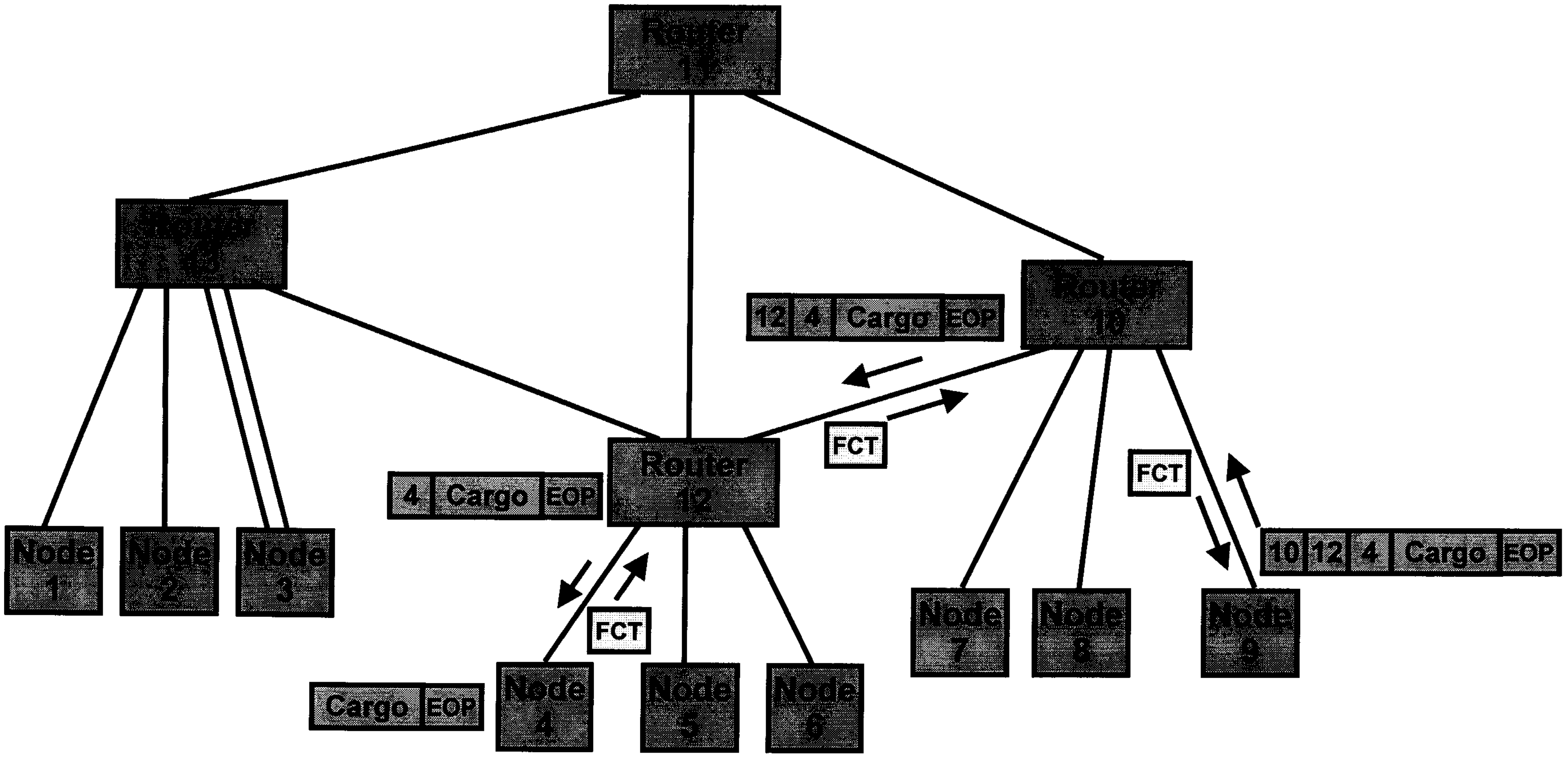
- **Advantages/Disadvantages of the Fault Tolerant IEEE 1394/I2C COTS Avionics Bus**
  - **Advantages**
    - No Modification to the IEEE 1394 and I2C bus COTS standards
    - Only one set of IEEE 1394/I2C buses has to be powered on. This is critical for power limited deep space missions
    - Capable of tolerating at least 3 faults
  - **Disavantages**
    - Complex design: 2 IEEE 1394 buses and 2 I2C buses
    - The I2C bus cannot handle the workload of the IEEE 1394 bus during fault recovery. This results in the necessity of the backup IEEE 1394/I2C bus set

Need: A more capable bus in the Design Diversity layer to assist the IEEE 1394 bus recovery and handle the workload during fault recovery

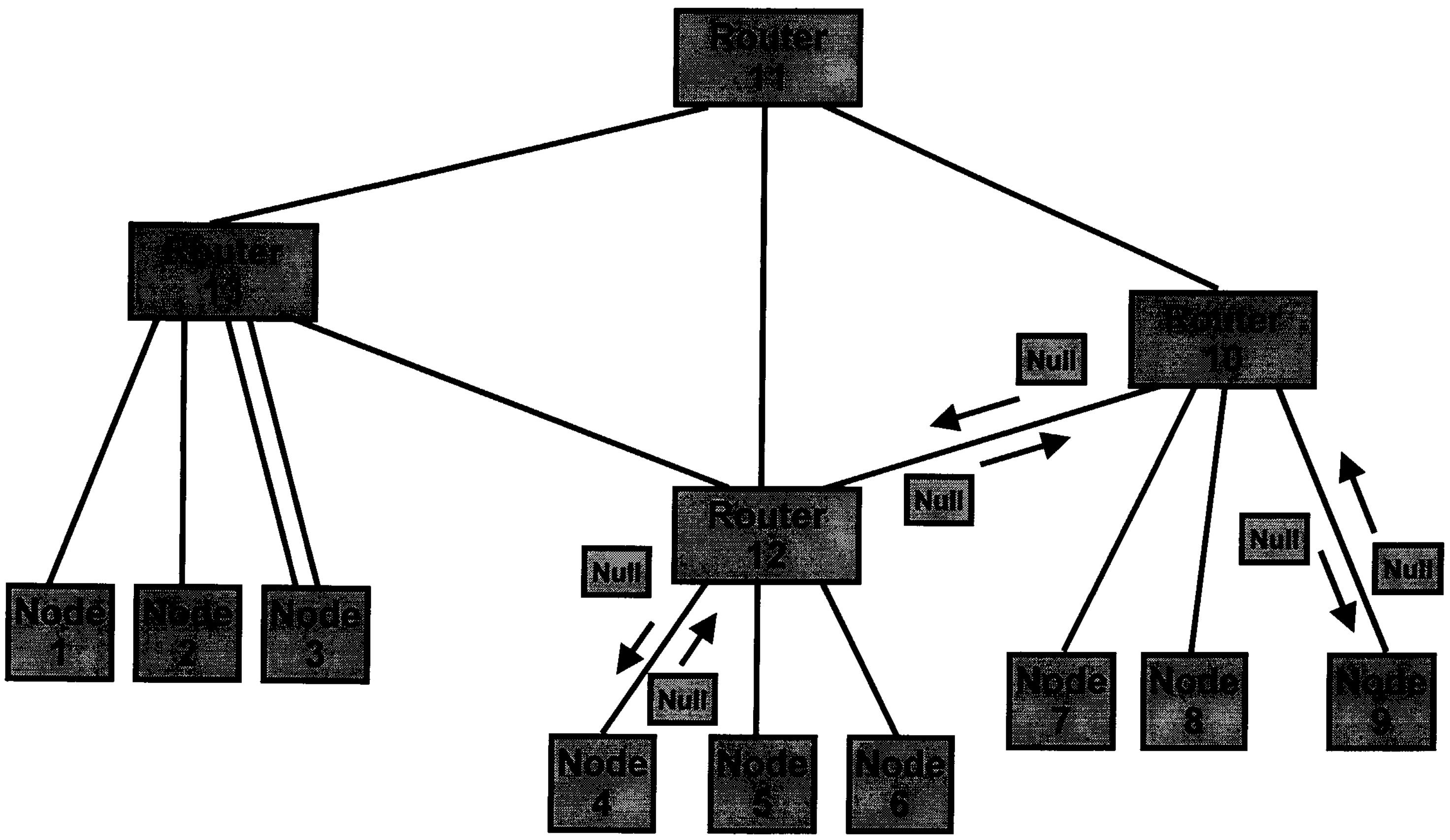
Alternative Backup Bus: SpaceWire

- Derived from the IEEE 1355 Bus
- Being proposed as one of the buses for the Spacecraft On-board Interface (SOIF) Standard
- Data rate: 2 Mbps to 400 Mbps. Some projects at JPL has achieved 10 Mbps data rate
- **Free topology**
- Six levels of protocols defined: physical, signal, character, exchange, packet, and network
- Physical layer is almost identical to IEEE 1394 bus
  - Data and Strobe signals
  - Use Low Voltage Differential Signal (LVDS) bus drivers
  - Full duplex as oppose to half duplex in IEEE 1394
- Simple higher level protocols

SpaceWire Bus Routing



Maintenance of SpaceWire Connection



Summary of Error Handling in SpaceWire

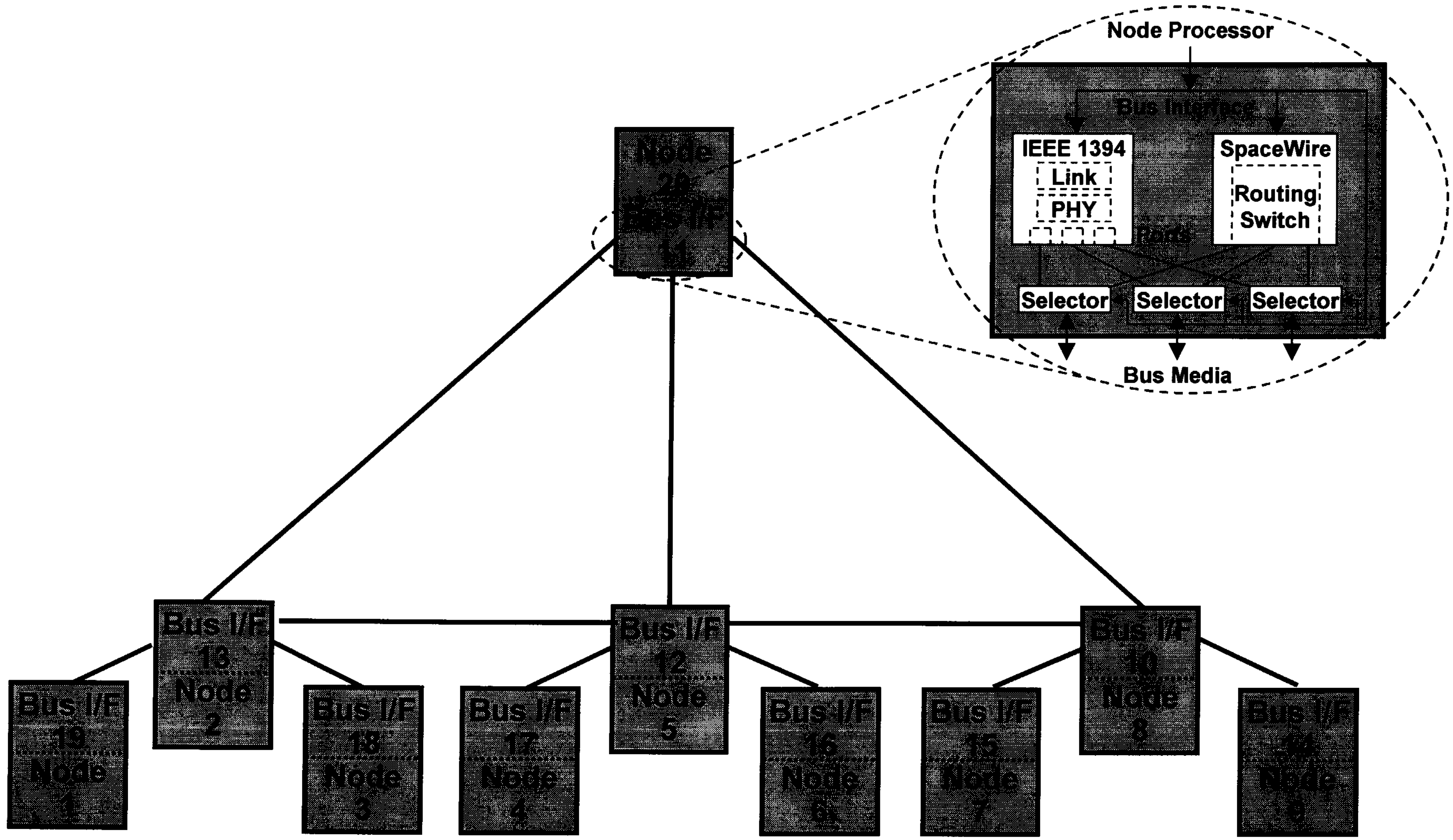
- Character Level
  - Detection: Parity Bit
  - Recovery: Both ends of the link shall be reset and re-initialised to recover character synchronisation and flow control status.
- Exchange Level
  - Detection: Disconnect errors, parity errors, escape errors, credit errors and character sequence errors
  - Recovery: lagged up to the network level as a link error
- Network Level:
  - Detection: Link Error, EEP (Error End of Packet) received, Invalid destination address
  - Recovery: If the error is detected at a source/destination node then the error shall be flagged up to the application level. If the error is detected within a routing switch, then the error may be flagged to a pin on the routing switch device or to an internal status register within the routing switch.

Comparison of SpaceWire and IEEE 1394 Buses

- IEEE 1394 Bus has bandwidth of 100, 200, 400 Mbps. SpaceWire has bandwidth 2 to 400 Mbps.
- IEEE 1394 Bus adopts a tree topology. SpaceWire has free topology.
- IEEE 1394 Bus has more capable high level protocol (e.g. bus initialization). SpaceWire has simpler high level protocol.
- IEEE 1394 Bus implementation is more complicate than the SpaceWire.
- IEEE 1394 Bus supports guaranteed bandwidth and bounded latency. SpaceWire does not support either.
- IEEE 1394 Bus supports multi-cast. SpaceWire is not capable of multi-cast.

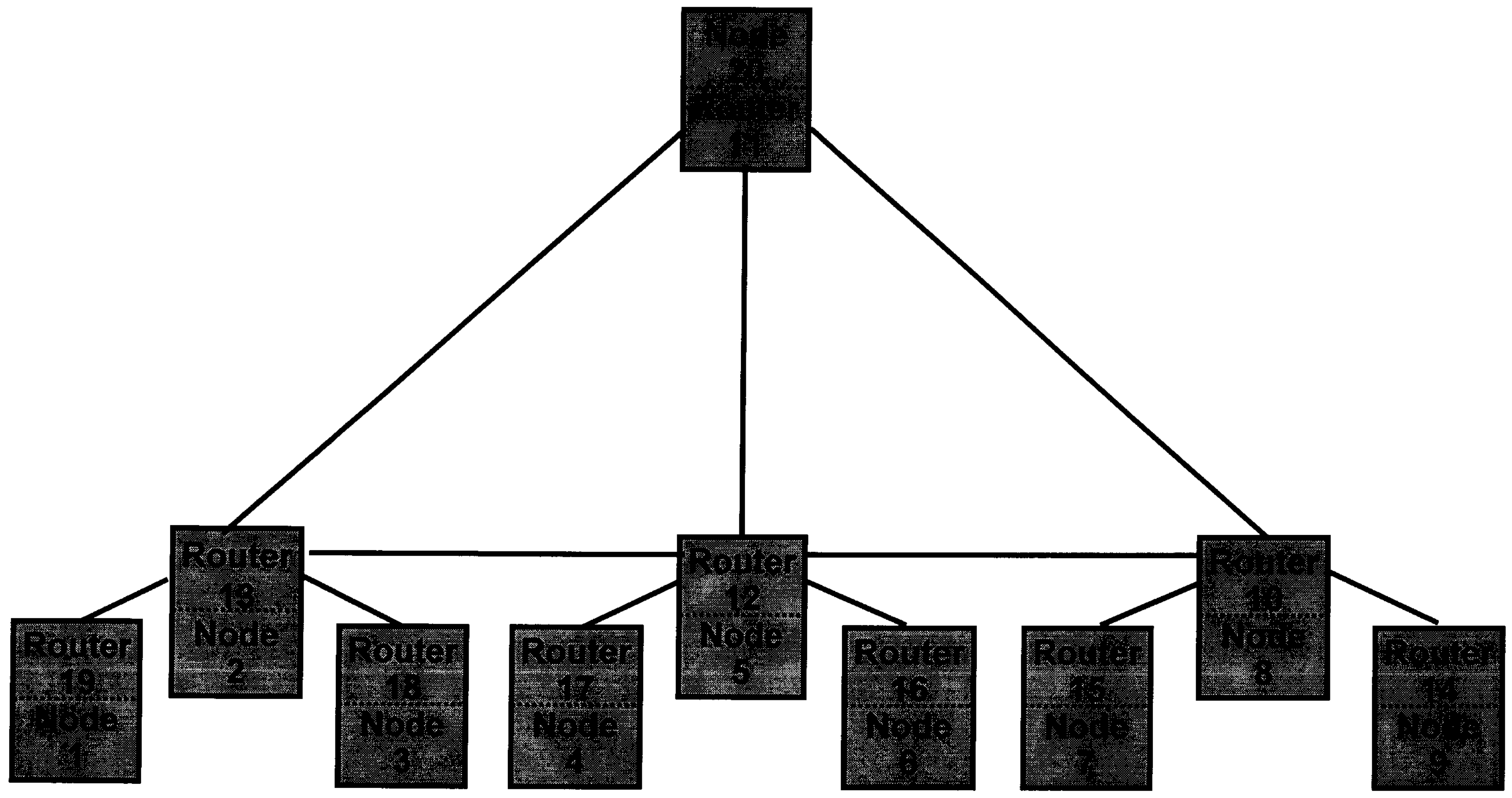
Conclusion: they are complementary and can be combined

Combining IEEE 1394 and SpaceWire at the Physical Layer



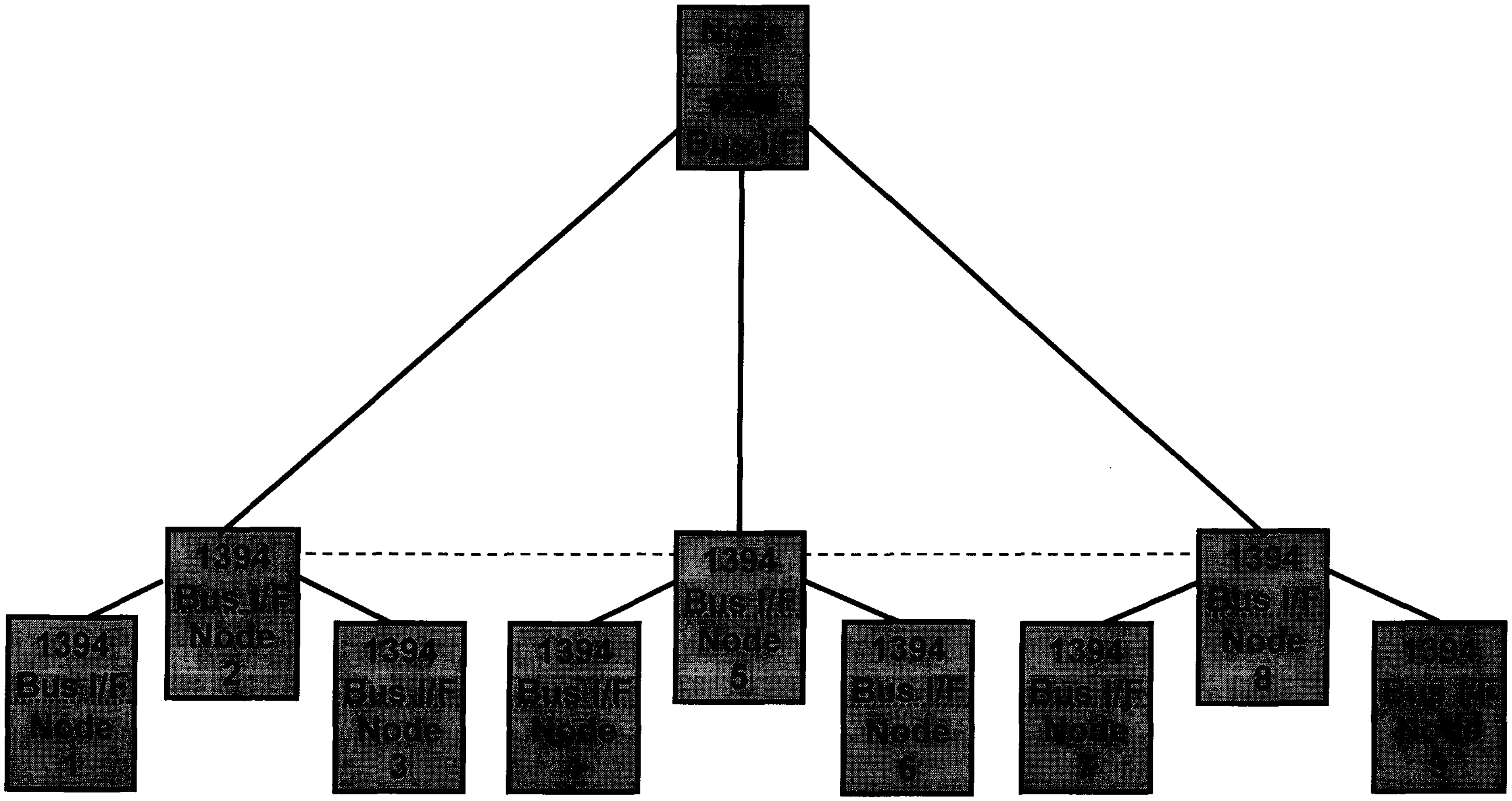
Operations of IEEE 1394/SpaceWire Bus

At system startup, all bus interfaces default to the SpaceWire mode since its bus initialization process is simpler



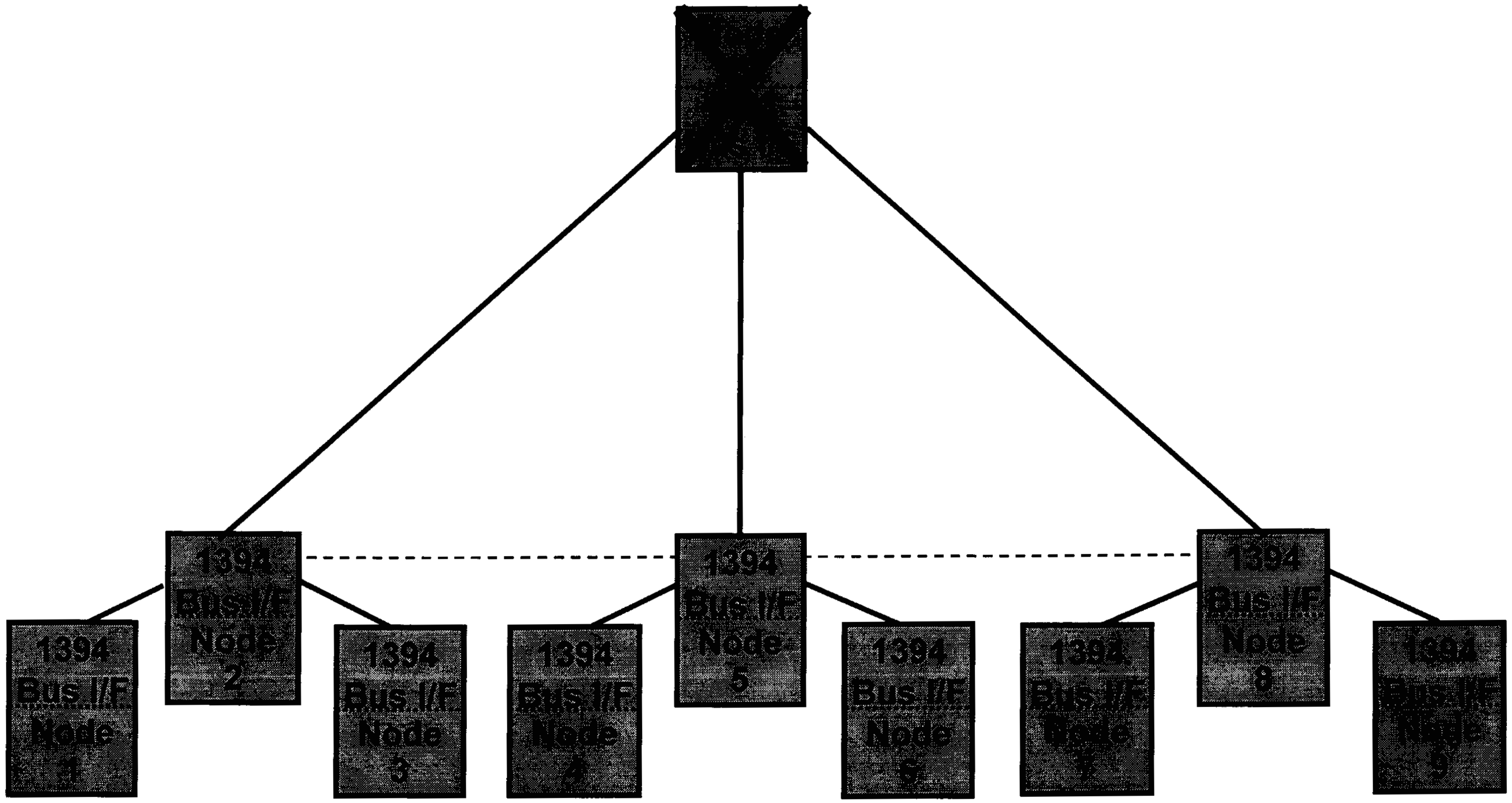
Operations of IEEE 1394/SpaceWire Bus

After the system is stabilized, an embedded tree will be selected to form the IEEE 1394 Bus for higher performance and guaranteed bandwidth



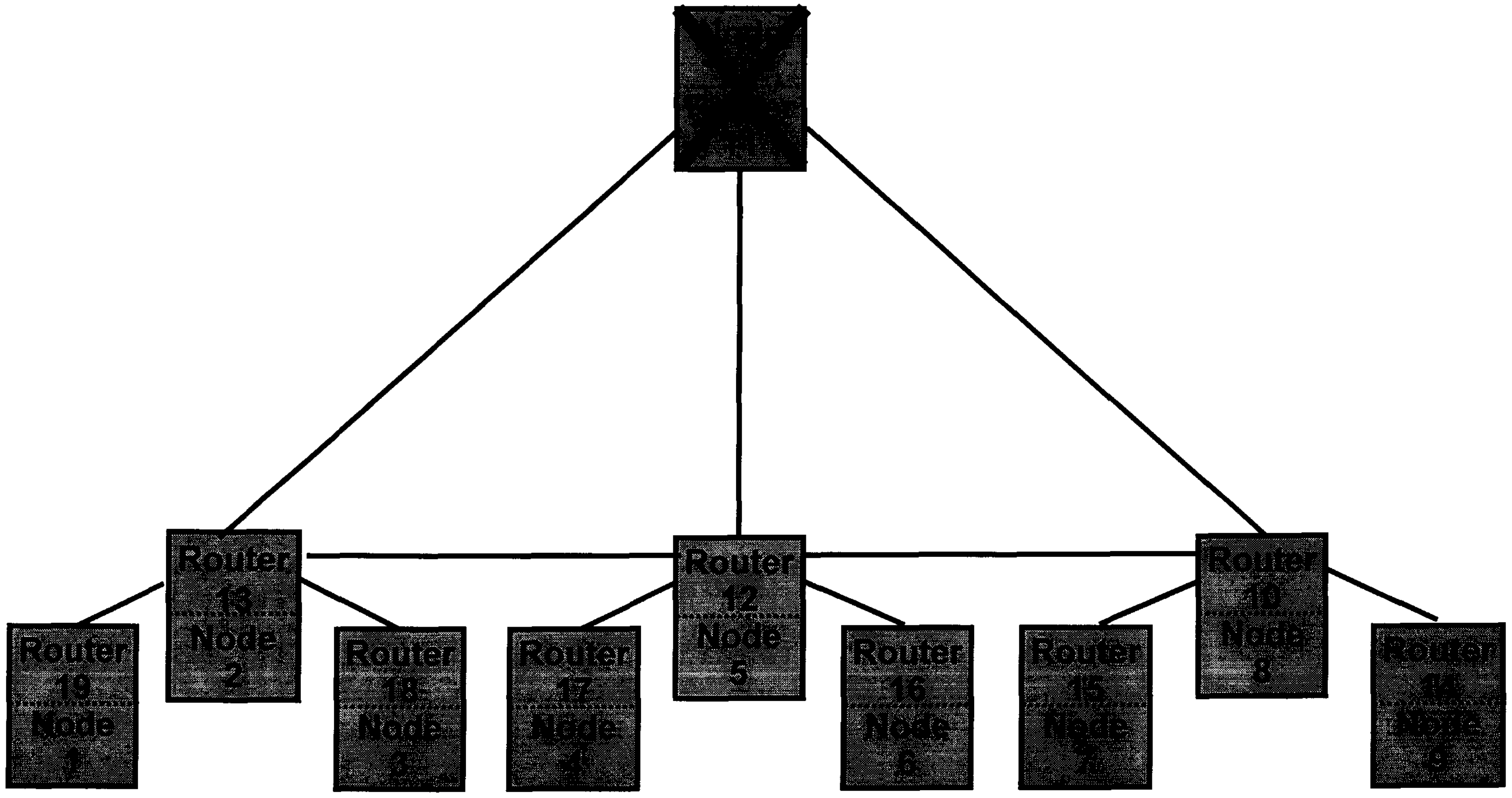
IEEE 1394/SpaceWire Bus Failure

Failure detected by the IEEE 1394 bus by polling or heartbeat



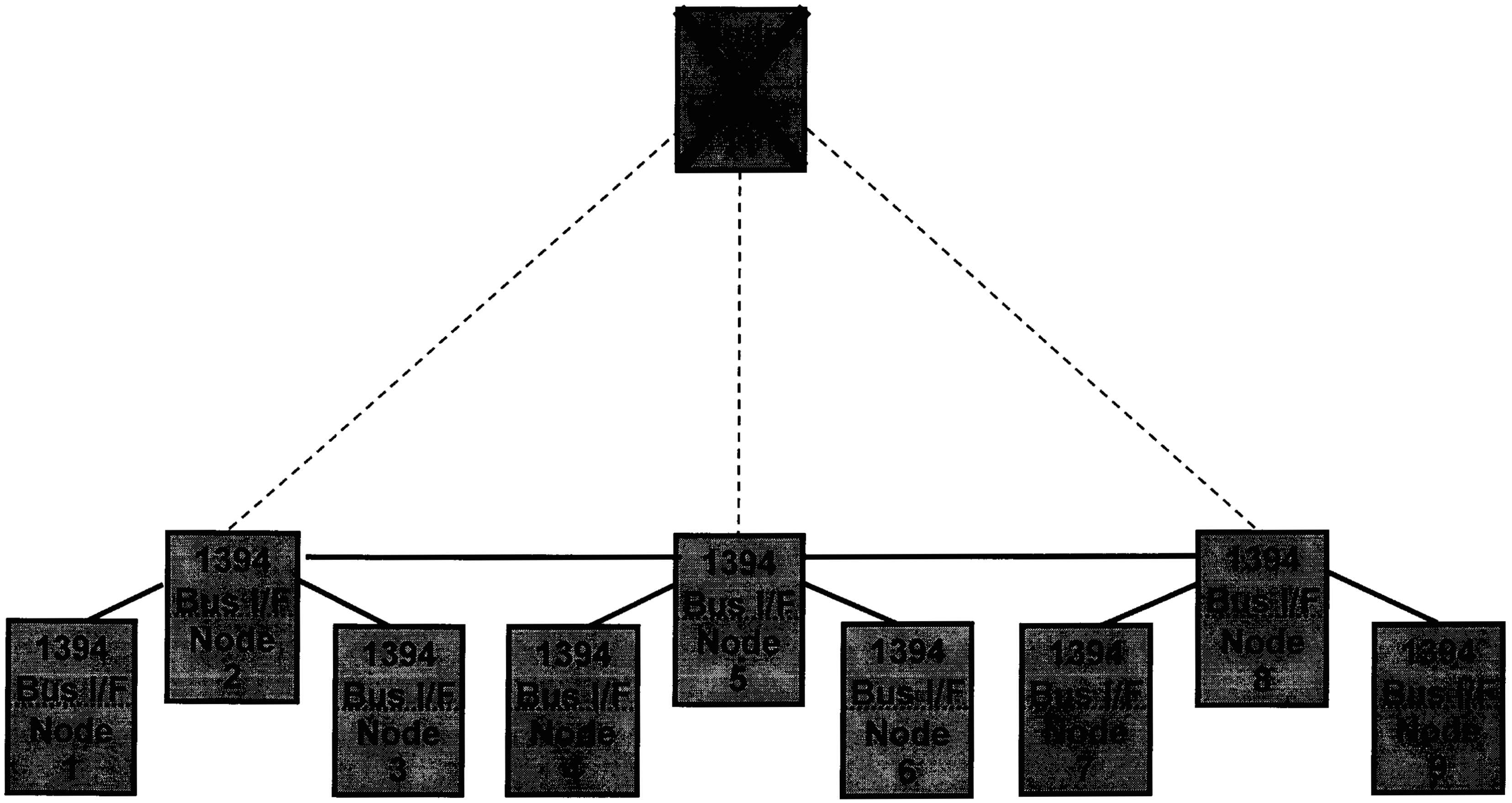
IEEE 1394/SpaceWire Bus Fault Recovery

Upon the detection of a node failure, all bus interfaces fall back to the SpaceWire mode and exchange messages to locate the failed node



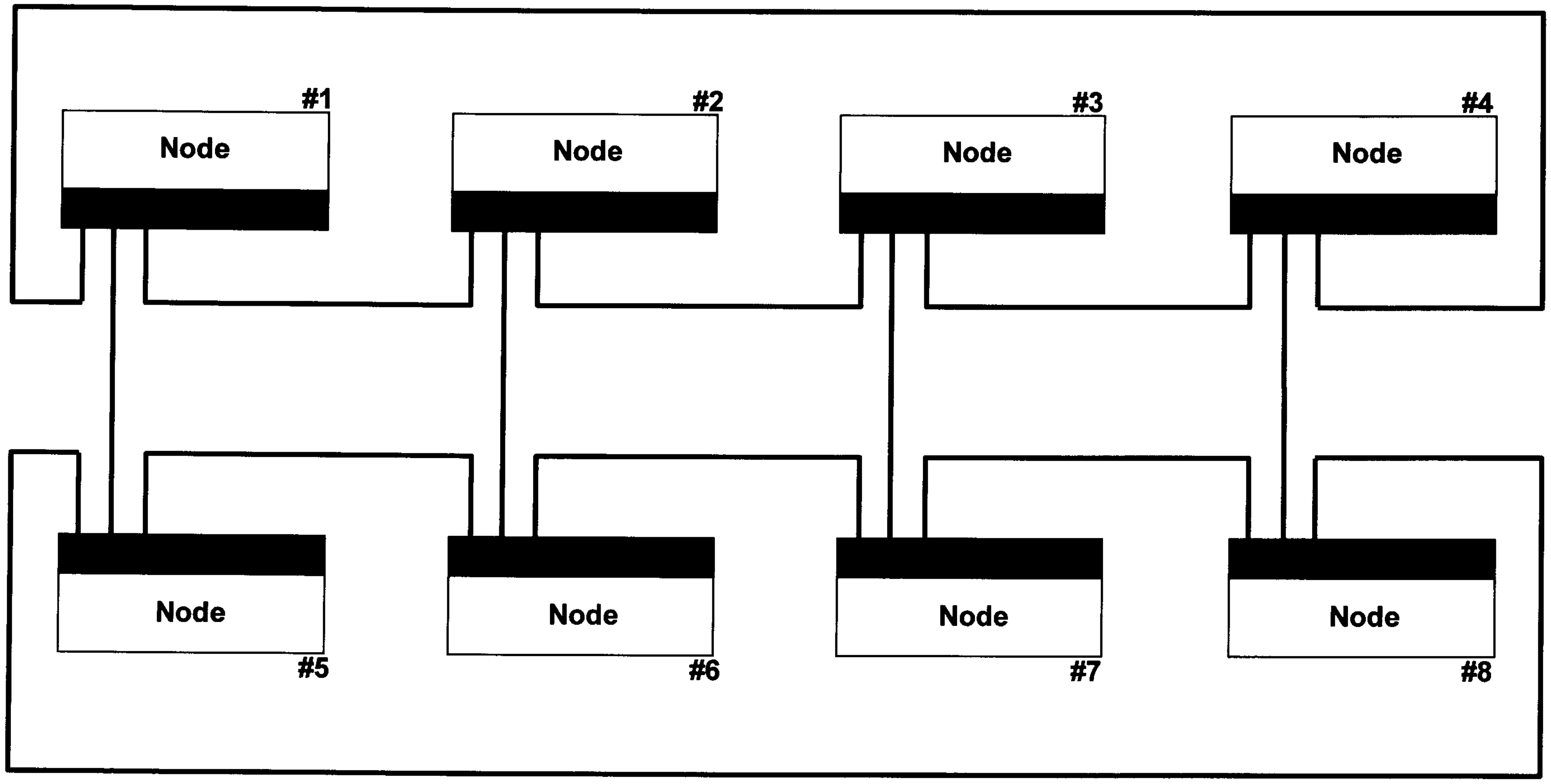
IEEE 1394/SpaceWire Bus Fault Recovery

After the failed node is isolated, another embedded tree will be selected to form the IEEE 1394 Bus again



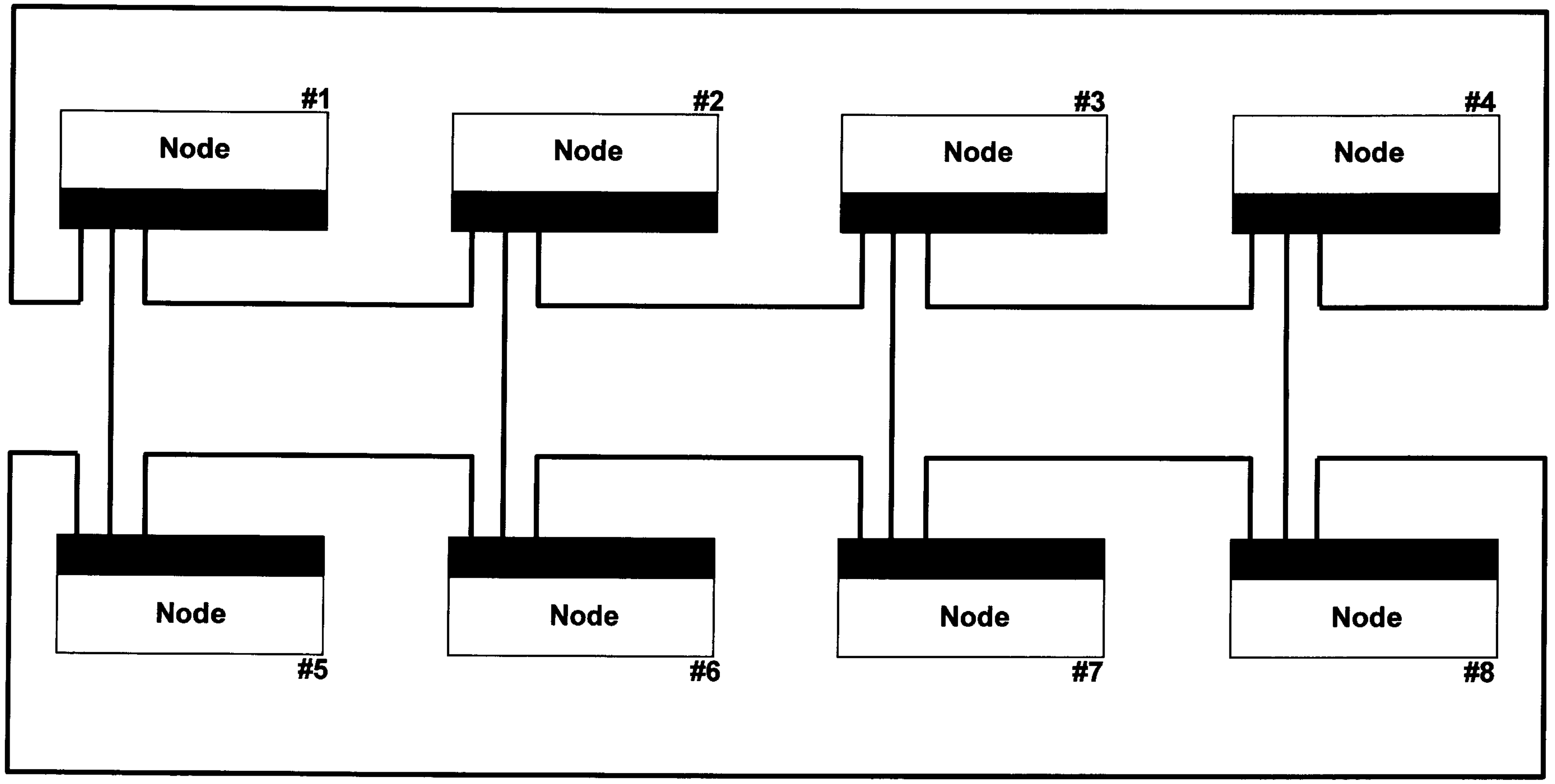
Implementation of the Dual Stack Tree Topology with IEEE 1394/SpaceWire Bus

At System Startup



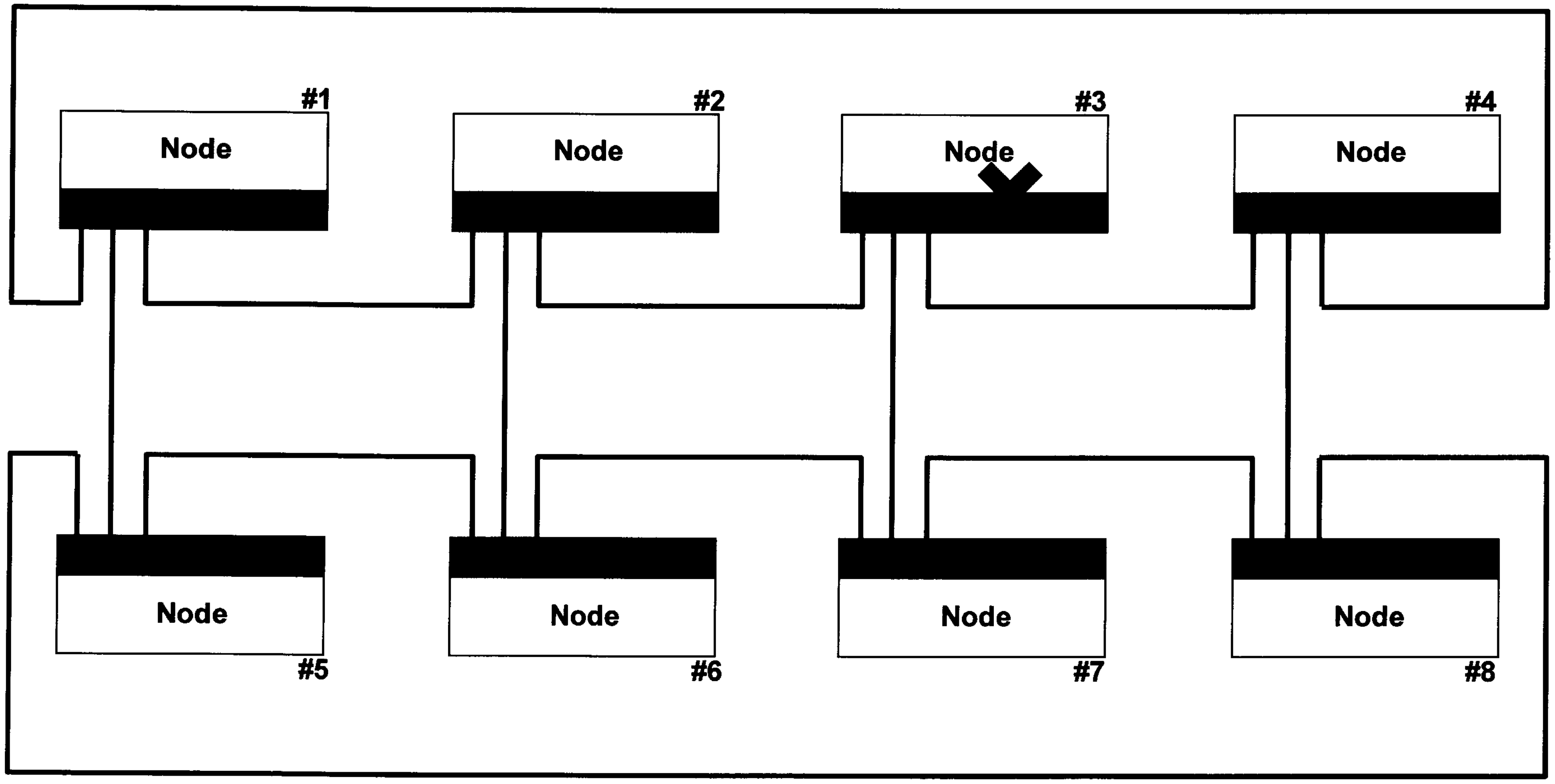
Implementation of the Dual Stack Tree Topology with IEEE 1394/SpaceWire Bus

Normal System Operation



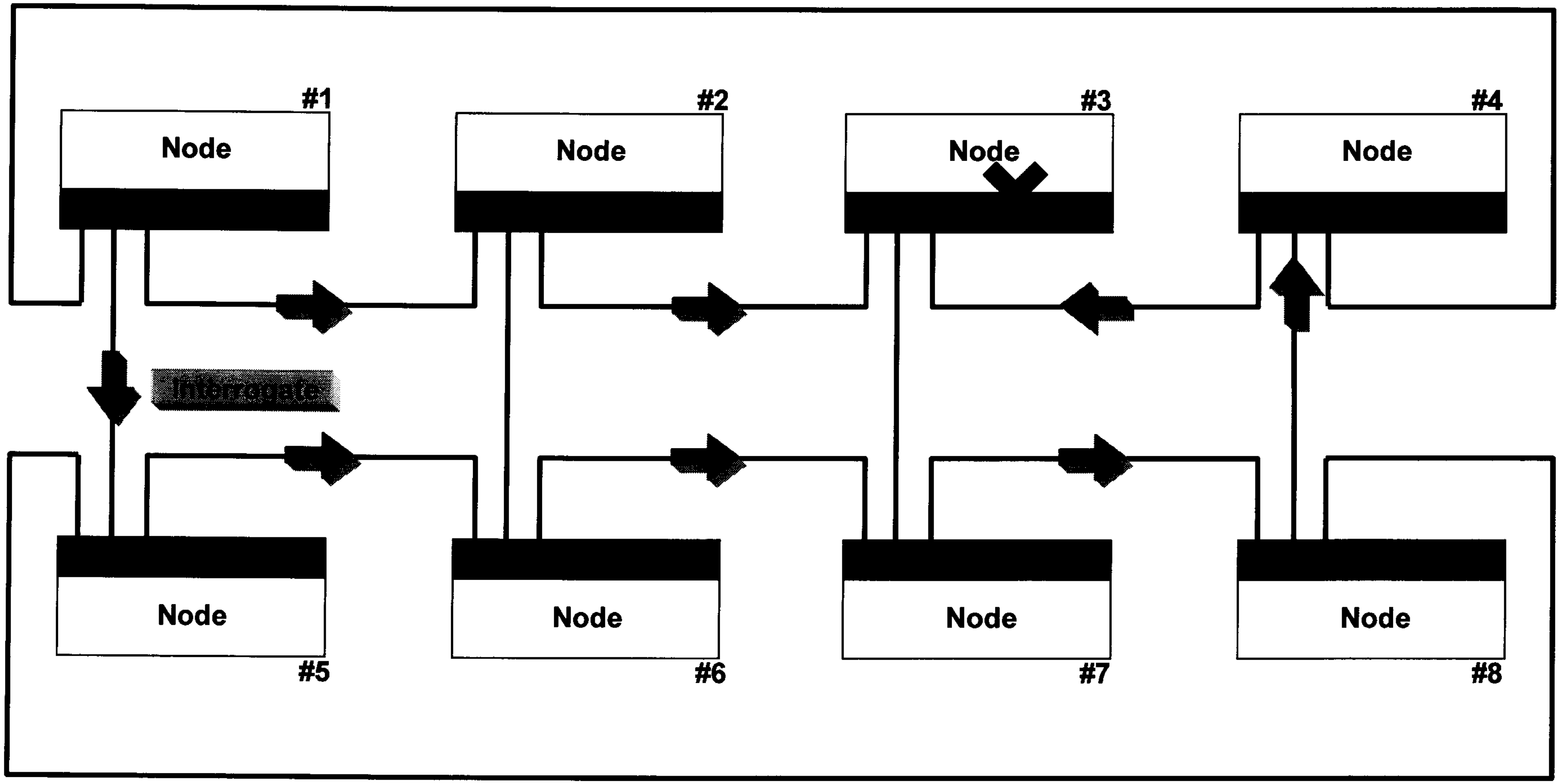
Fault Recovery of the Dual Stack Tree IEEE 1394/SpaceWire Bus

Normal System Operation



Fault Recovery of the Dual Stack Tree IEEE 1394/SpaceWire Bus

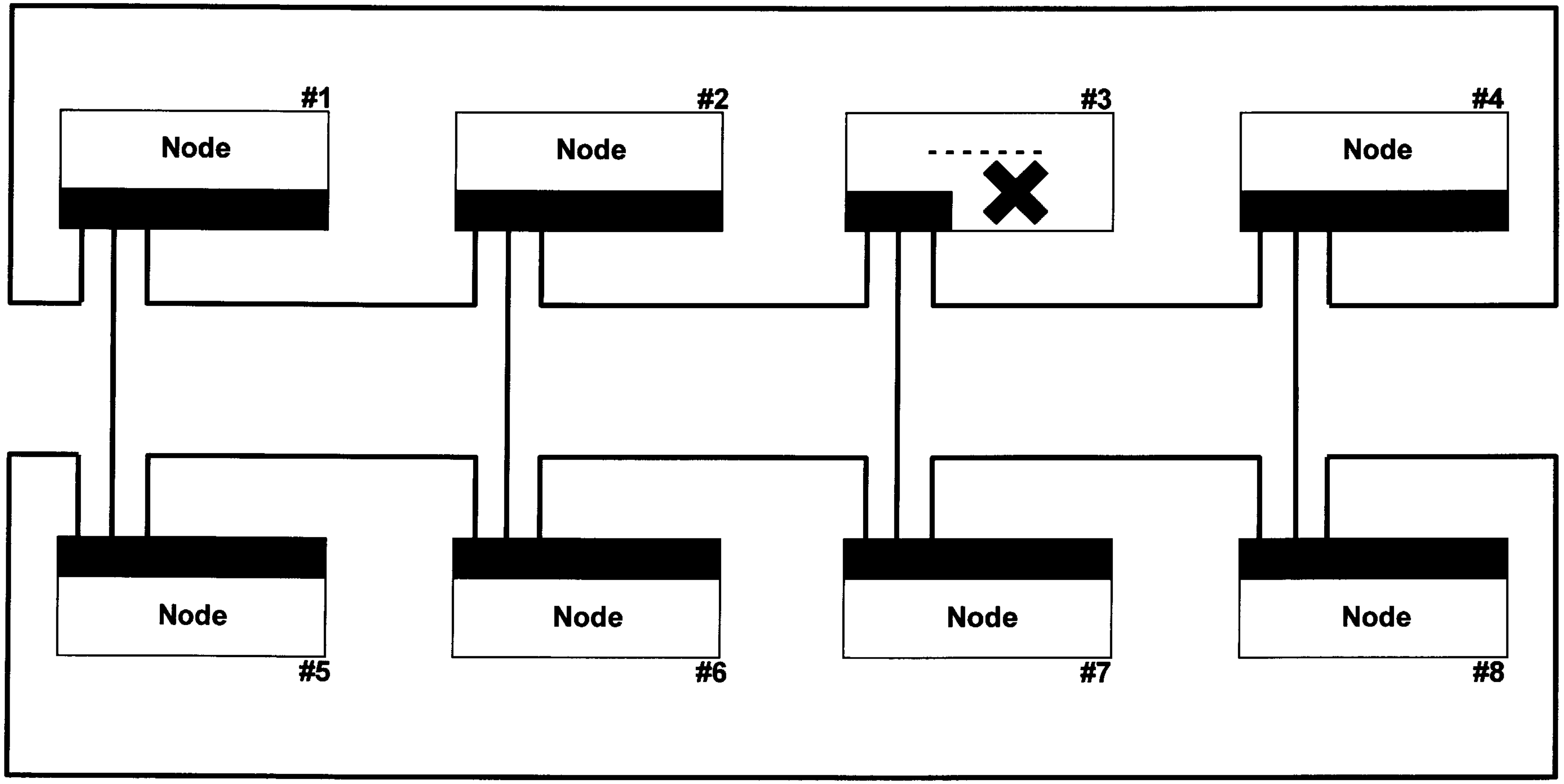
Upon Node Failure: System falls back to SpaceWire



Note: Since SpaceWire has considerable high bandwidth, most normal operation except very large data block transfer can continue during fault recovery

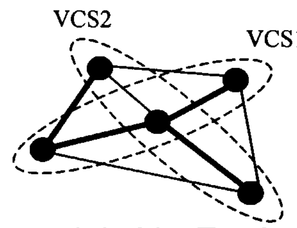
Fault Recovery of the Dual Stack Tree IEEE 1394/SpaceWire Bus

After Fault Recovery

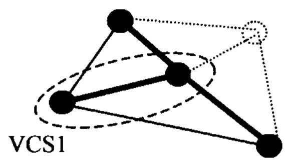


Effectiveness of the IEEE 1394/SpaceWire Avionics Bus Fault Tolerance

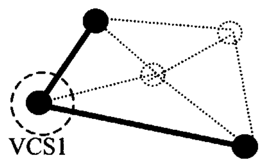
- ***Definition 1:*** A *vertex-cut-set (VCS)* is a minimal set of vertices in a graph, the removal of which will partition the graph, whereas the removal of any proper subset of which will not partition the graph.
- ***Definition 2:*** An *edge-cut-set (ECS)* is a minimal set of edges in a graph, the removal of which will partition the graph, whereas the removal of any proper subset of which will not partition the graph.
- ***Lemma 1:*** A graph is connected if and only if it contains a spanning tree.
- ***Theorem 1:*** If G is a connected graph corresponding to the initial connections established by the SpaceWire bus, then the combined IEEE-1394/SpaceWire bus network can tolerate at least n – 1 node (vertex) failures or m – 1 link (edge) failures, where n and l are the sizes of the vertex-cut-set and edge-cut-set of G, respectively.
- ***Proof:*** See Paper
- ***Example:***



(a): No Fault
Size of VCS1 = 3
Size of VCS2 = 3

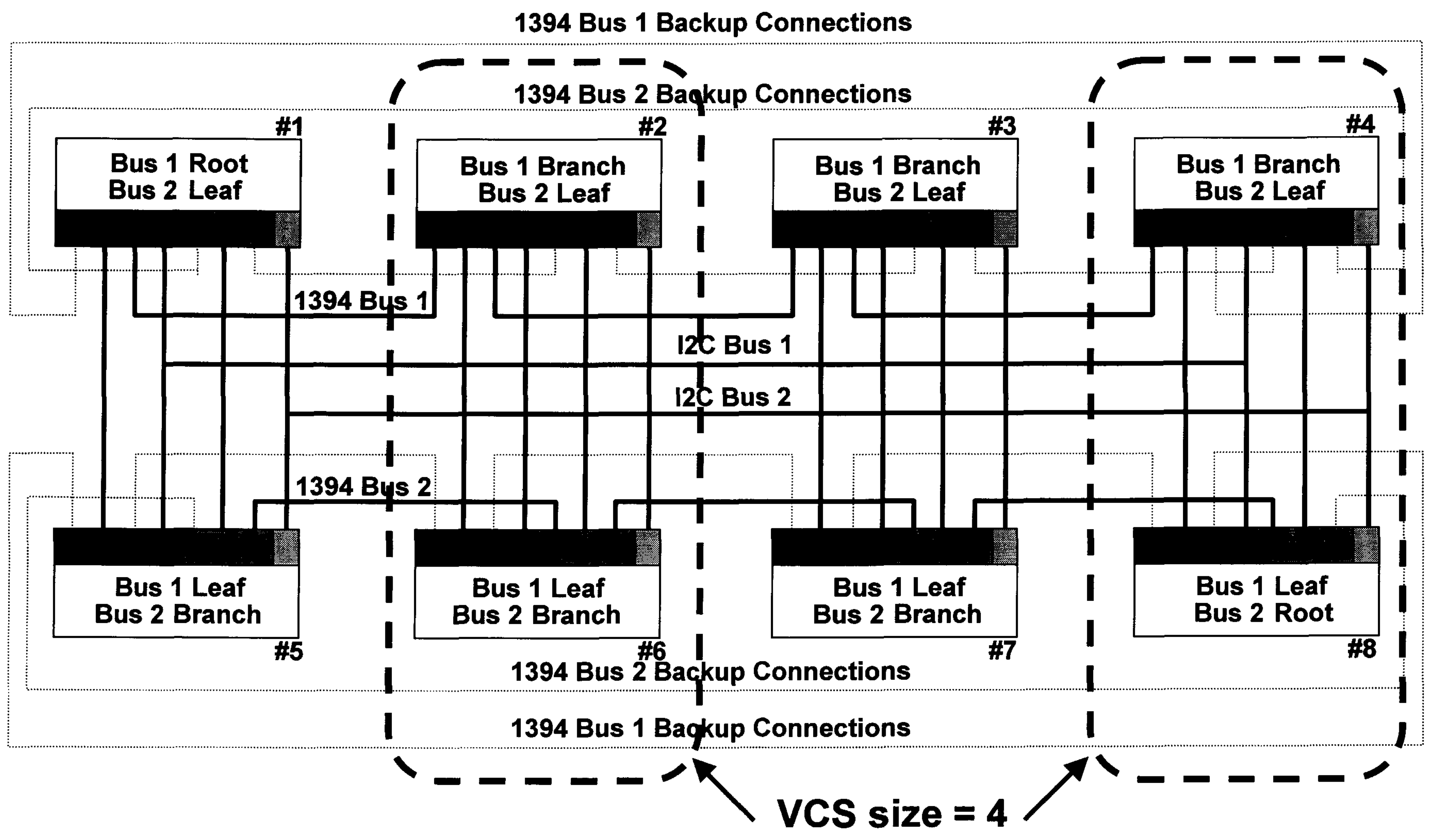


(b): One Fault
Size of VCS1 = 2



(c): Two Faults
Size of VCS1 = 1

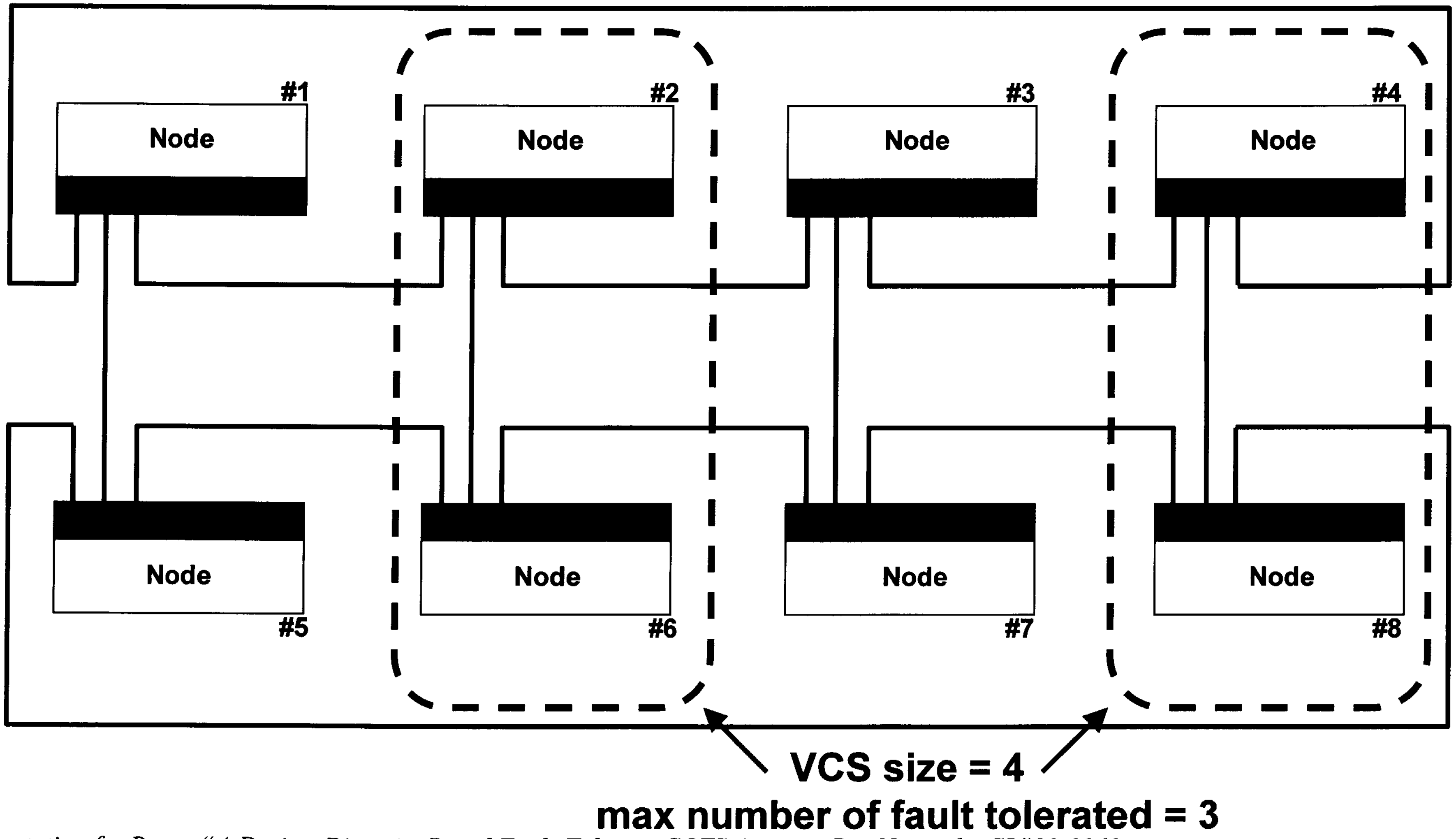
Applying the Theorem to the IEEE 1394/I2C Bus



max number of fault tolerated = 3

Compare to the Effectiveness of the IEEE 1394/SpaceWire Bus

- The IEEE 1394/SpaceWire bus architecture is as effective as the IEEE/I2C bus architecture in fault tolerance



Conclusion

- The IEEE 1394 and SpaceWire have complementary advantages: the IEEE 1394 bus has more powerful protocols and guaranteed bandwidth, while the SpaceWire has more flexible topology
- The IEEE 1394 and SpaceWire has very similar physical layer protocol and therefore can share the same bus media
- Due to the high performance of the SpaceWire, there is no need for a redundant IEEE 1394 to maintain system operation or I2C bus to support the fault recovery of the IEEE 1394 Bus.
- Consequently, the combined IEEE 1394/SpaceWire bus architecture has much simpler connectivity than the original IEEE/I2C bus architecture
- The IEEE 1394/SpaceWire bus architecture has the same effectiveness in fault tolerance as the IEEE/I2C bus architecture

Future Work

- Develop algorithms to establish the optimal topology that has minimum number of connections but has the maximum size *vertex-cut-set* and *edge-cut-set*
- Evaluate the complexity of the IEEE 1394/SpaceWire bus interface with hardware implementation
- Evaluate the complexity of the software needed by the fault tolerance, especially the switching between two networks
- Perform fault injection to evaluate the fault tolerance capability of the bus architecture

Guarded Software Upgrade

- Another Dependable Computing Research at JPL
- Collaboration with Dr. Ann Tai and Dr. Kam Tso in IA Tech
- Motivation:
  - Onboard evolvability of embedded avionics systems has been increasingly viewed as the property crucial to the survival of a spacecraft in a long-life deep-space mission.
  - There have been cases in which unprotected software upgrades caused severe and costly damage to space missions and highly-available systems (e.g., First US space shuttle aborted launch, ARIANE-5 failure, MCI 10-day outage).
  - Experiences show that it is impossible to predict, during ground testing prior to uploading, all possible onboard conditions in an unsurveyed deep-space environment, and thus an upgraded embedded software component can never be guaranteed to have ultra-high reliability.
- Many Ground Applications Such As Network Server Software Upgrade

Guarded Software Upgrade: Another Dependable Computing Research at JPL

Necessity of Onboard Guarded Software Upgrading (GSU)

- Onboard evolvability of embedded avionics systems has been increasingly viewed as the property crucial to the survival of a spacecraft in a long-life deep-space mission.
- There have been cases in which unprotected software upgrades caused severe and costly damage to space missions and highly-available systems (e.g., first US space shuttle aborted launch, ARIANE-5 failure, MCI 10-day outage).
- Experiences show that it is impossible to predict, during ground testing prior to uploading, all possible onboard conditions in an unsurveyed deep-space environment, and thus an upgraded embedded software component can never be guaranteed to have ultra-high reliability.

Guarded Software Upgrade

GSU Framework

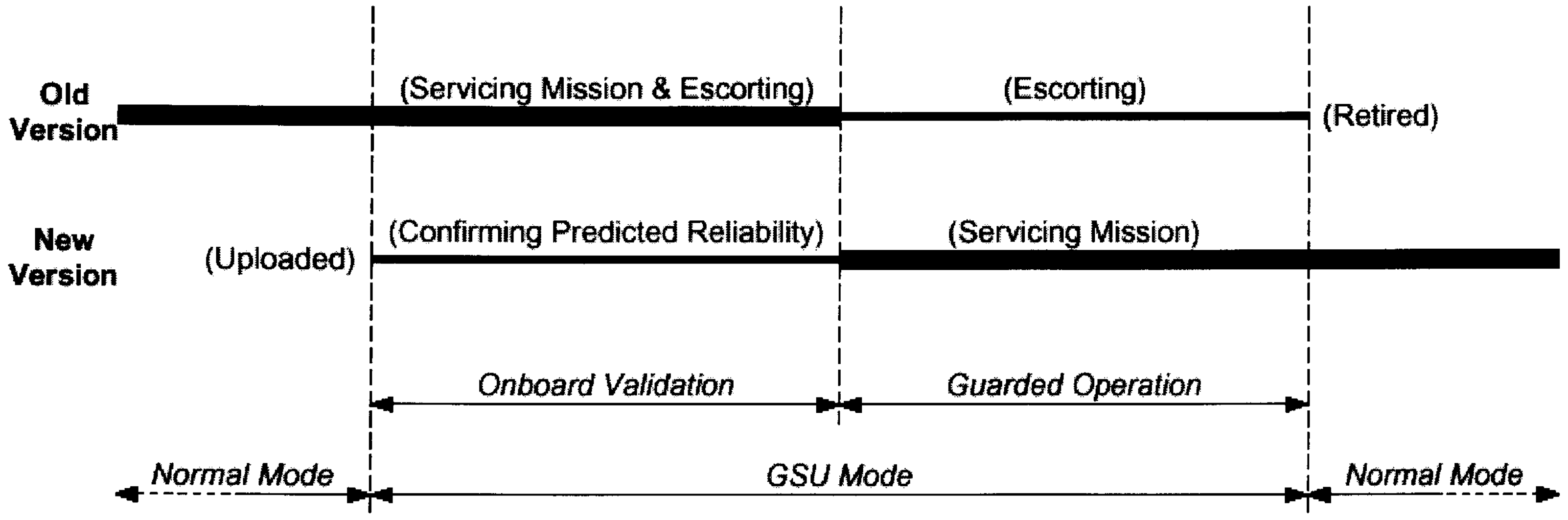
Application system considered: X2000 distributed architecture for multiple long-life deep-space missions.

Goal: To enhance system reliability when an application software component undergoes an onboard upgrade, with low development cost and low performance cost.

Means: Exploiting inherent/non-dedicated system resource redundancies (that will be available to us during onboard upgrading) and a message-driven confidence-driven (MDCD) approach.

Guarded Software Upgrade

GSU Methodology



Guarded Software Upgrade

A Message-Driven Confidence-Driven Fault Tolerance Protocol

Objective: To mitigate the effect of residual faults in an upgraded software component.

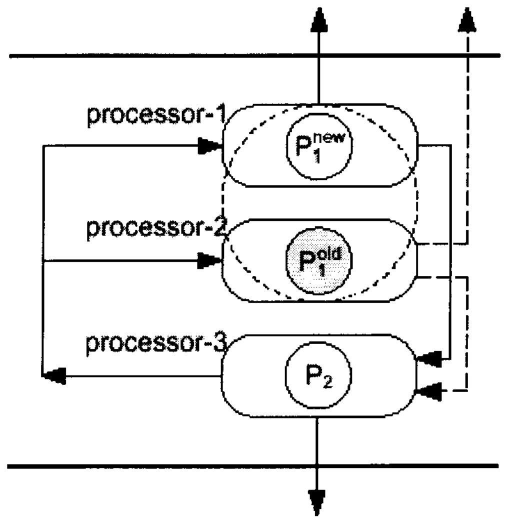
Approach: We take a crucial step in devising the error containment and recovery protocol by introducing the "confidence-driven" notion. This notion *complements* the message-driven (or "communication-induced") approach employed by a number of existing checkpointing protocols for tolerating hardware faults.

Guarded Software Upgrade

Dynamic Confidence-Driven Approach

1) We discriminate between the individual software components with respect to our confidence in their reliability, and

2) We dynamically adjust our confidence, at onboard execution time, in the processes corresponding to those software components, according to knowledge about potential process state contamination caused by errors in a low-confidence component and message passing.



Guarded Operation Stage

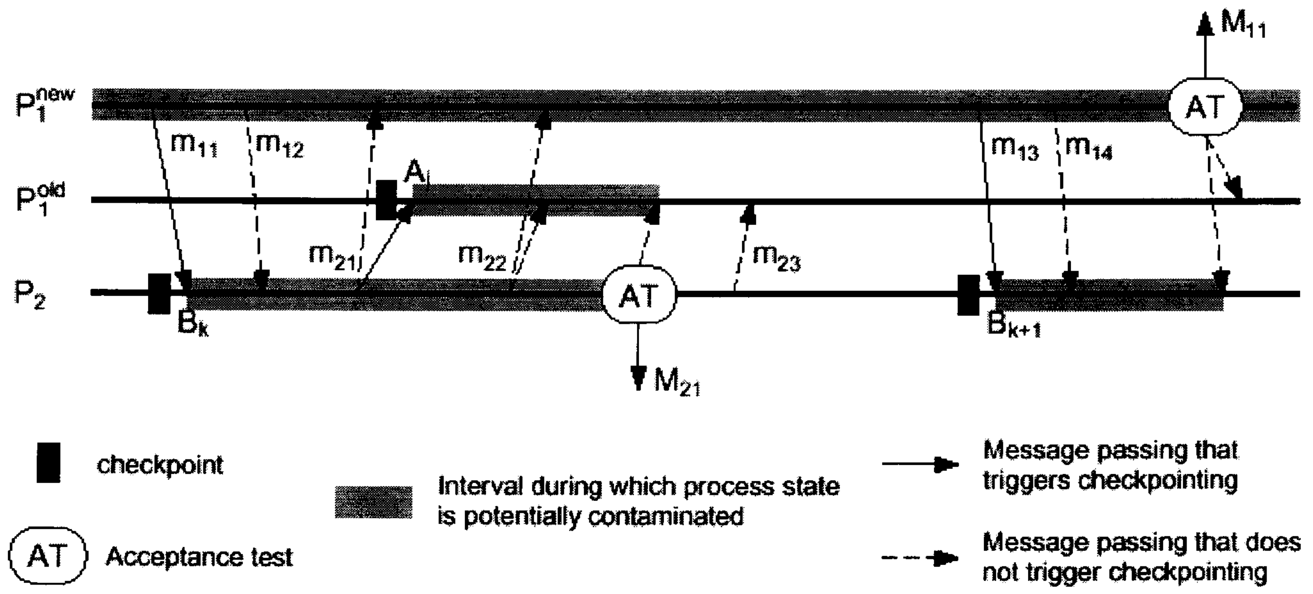
Guarded Software Upgrade

Definition: Potentially Contaminated Process State

By a "potentially contaminated process state," we mean

1) the state of the process corresponding to an initially identified low-confidence application software component (P_1^{new}), or

2) a process state that reflects the receipt of a not-yet-validated message that is sent by a process when its process state is potentially contaminated.

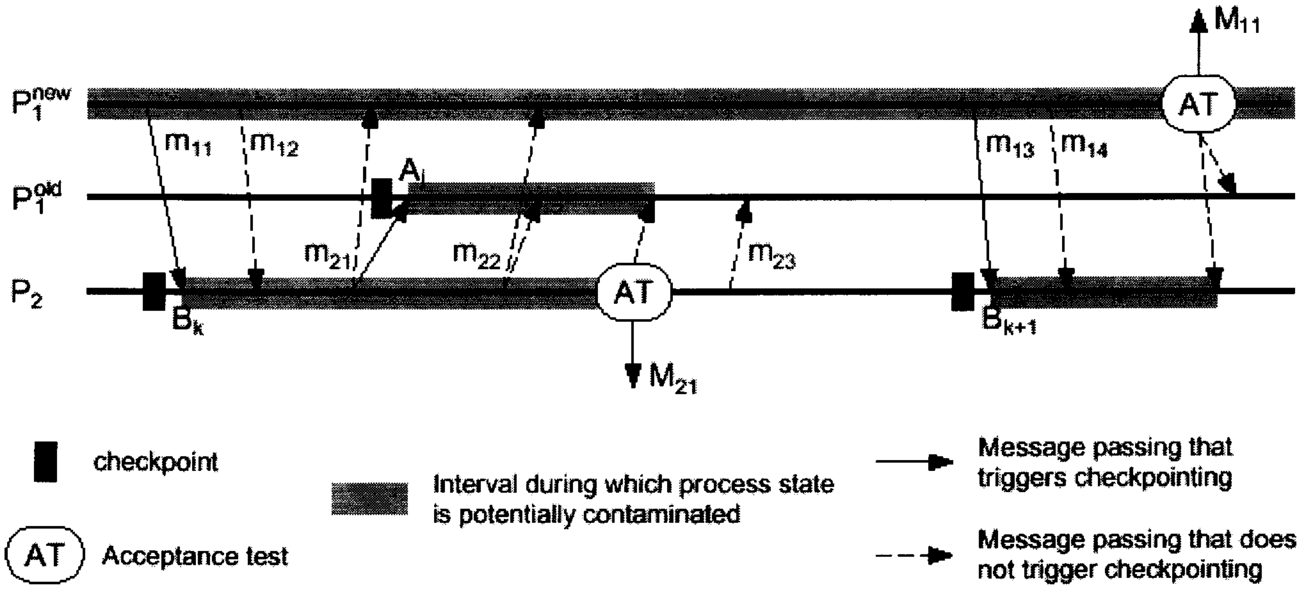

M_{11}
P_1^{new}
AT
m_{11}
m_{12}
m_{13}
m_{14}
A_i
P_1^{old}
m_{21}
m_{22}
m_{23}
P_2
AT
B_k
B_{k+1}
M_{21}
checkpoint
Interval during which process state is potentially contaminated
Message passing that triggers checkpointing
AT
Acceptance test
Message passing that does not trigger checkpointing

Guarded Software Upgrade

MDCD Error Containment

Checkpointing Rule: A process saves its state in a checkpoint if and only if the process is at the following point: immediately before its otherwise non-contaminated state becomes potentially contaminated.

AT-Based Validation Policy: We use an AT to validate a message that a process intends to send if and only if: 1) the message is an external message, and 2) the state of the sending process is potentially contaminated when the message is generated.

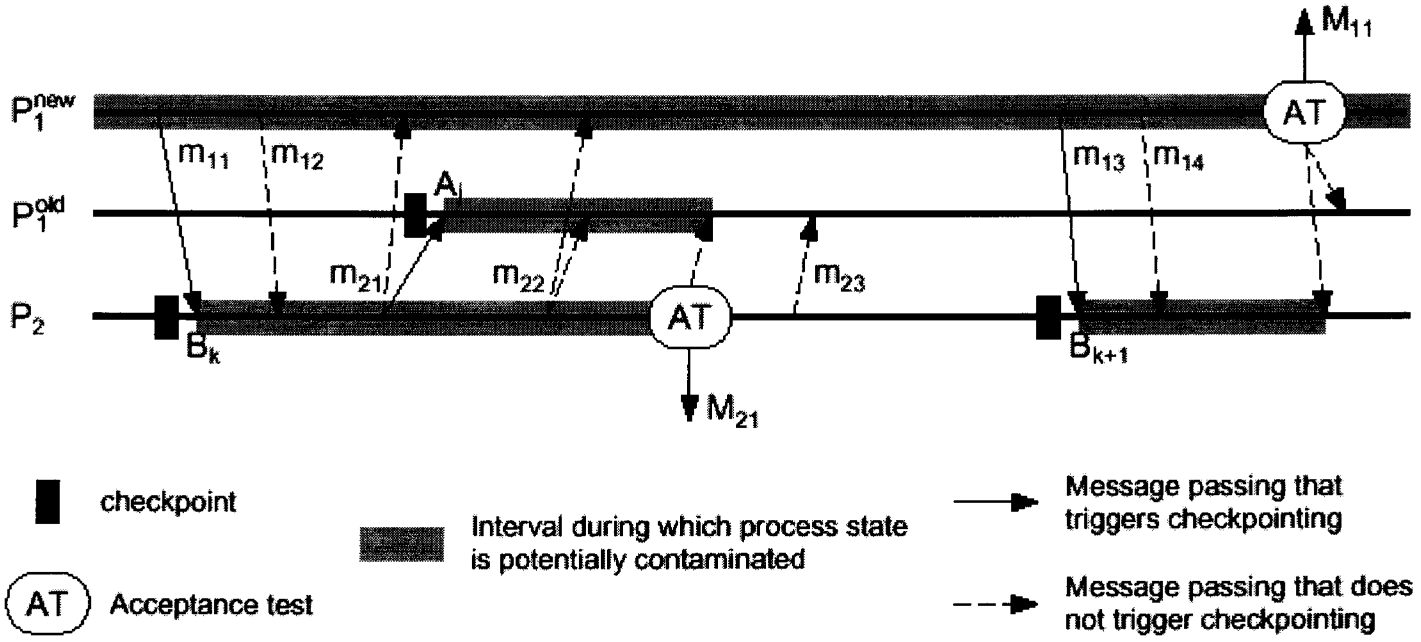


M_{11}
P_1^{new}
AT
m_{11}
m_{12}
m_{13}
m_{14}
A_i
P_1^{old}
m_{21}
m_{22}
m_{23}
P_2
AT
B_k
B_{k+1}
M_{21}
checkpoint
Interval during which process state is potentially contaminated
Message passing that triggers checkpointing
Message passing that does not trigger checkpointing
AT
Acceptance test

Guarded Software Upgrade

MDCD Error Recovery

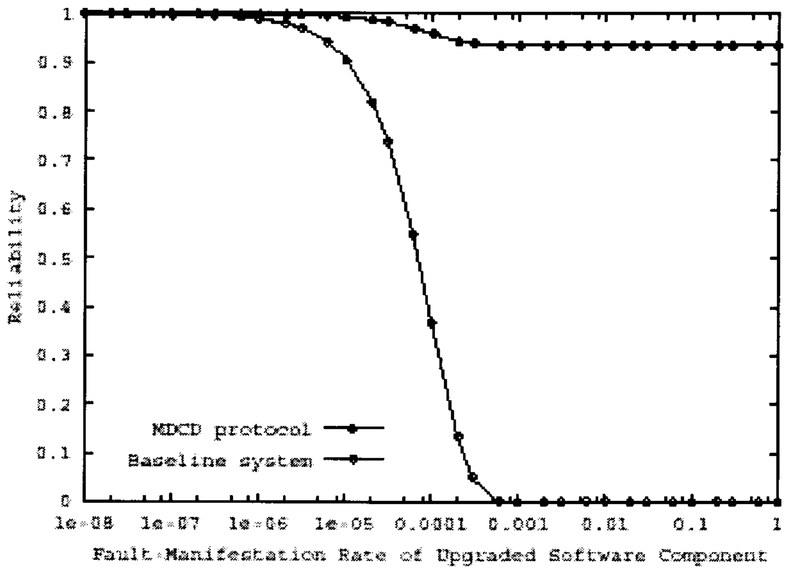
Since the error containment methods enable the interacting processes to maintain their knowledge about potential process state contamination and message validity, individual processes (P_1^{old} and P_2) are able to, upon error detection, make their own decisions on rollback (to the most recent checkpoint) or roll-forward, without requiring a costly global decision algorithm.



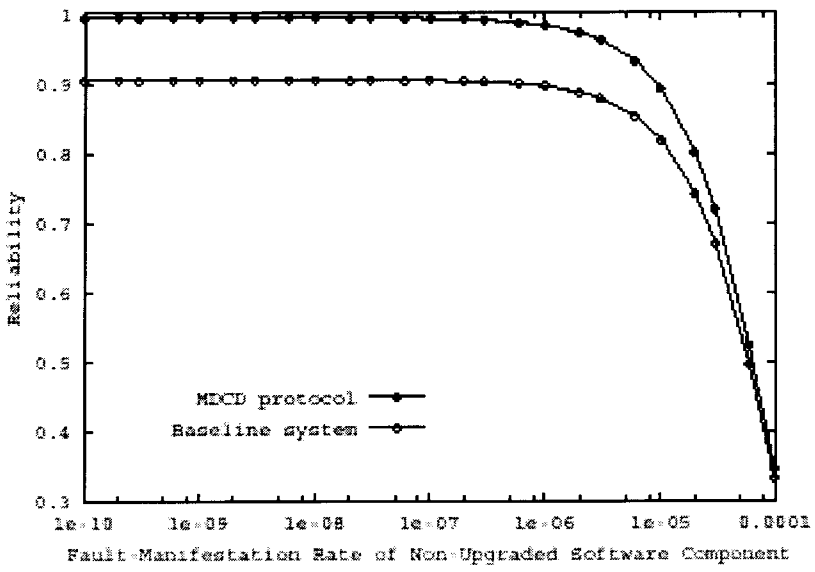
M11
P1 new
AT
m11
m12
m13
m14
P1 old
A_j
m21
m22
m23
P2
AT
B_k
B_k+1
M21
checkpoint
Interval during which process state is potentially contaminated
Message passing that triggers checkpointing
AT Acceptance test
Message passing that does not trigger checkpointing

Guarded Software Upgrade

Effectiveness with Respect to Reliability Gain



(a) As a Function of μ_{new}



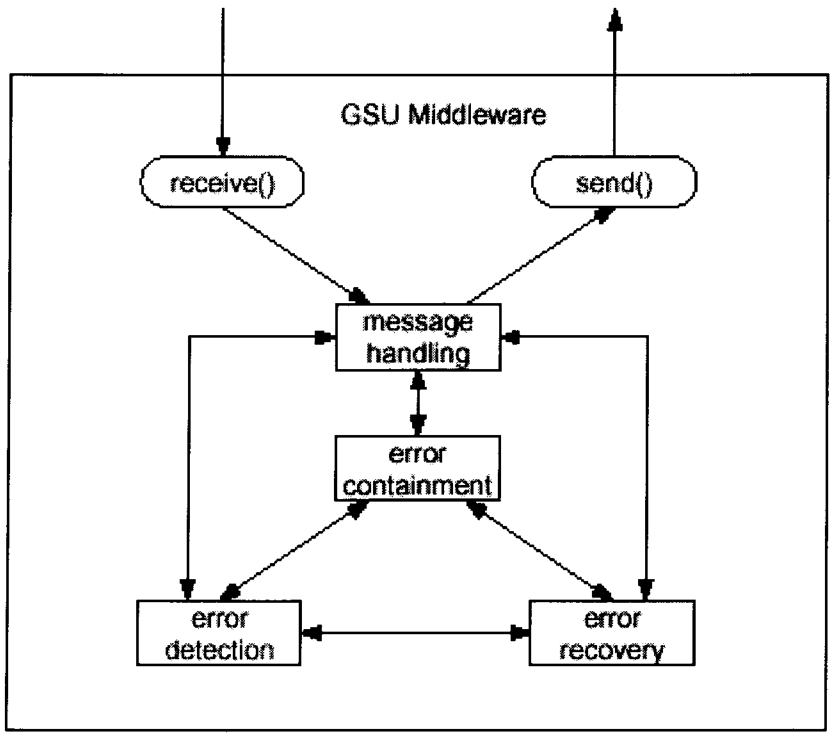
(b) As a Function of μ_{old}

JPL

Guarded Software Upgrade

Protocol Implementation: GSU Middleware

The dynamic nature of the MDCD protocol allows the error containment and recovery mechanisms to be transparent to the programmer and thus facilitates a middleware implementation.



Guarded Software Upgrade

Major Ingredients of GSU Middleware

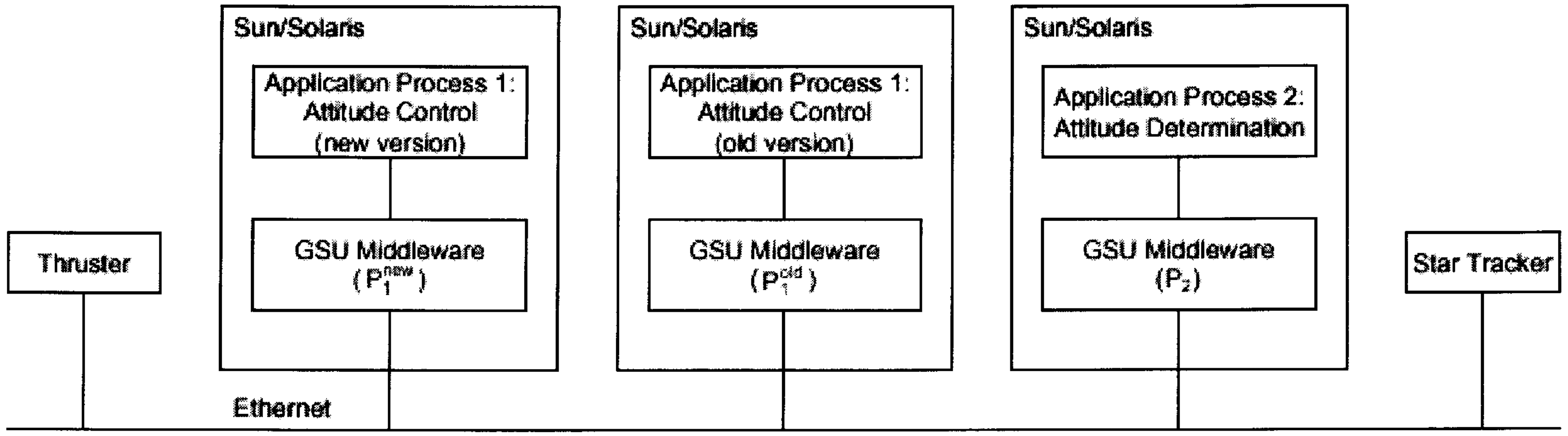
1) A set of *invocable services* that execute a message-sending or -receiving request from an application process, in a manner adaptive to
   - Confidence in the sender and/or receiving processes.
   - Criticality of the message (internal or external).
   - Nature of the GSU stage (onboard validation or guarded operation).

2) A collection of *MDCD modules* responsible for
   - Adjusting confidence in a process based on message-passing events, and
   - Making decisions on whether to take a checkpoint upon a message-passing event and whether to roll back or roll forward during error recovery.

JPL

Guarded Software Upgrade

GSU Testbed

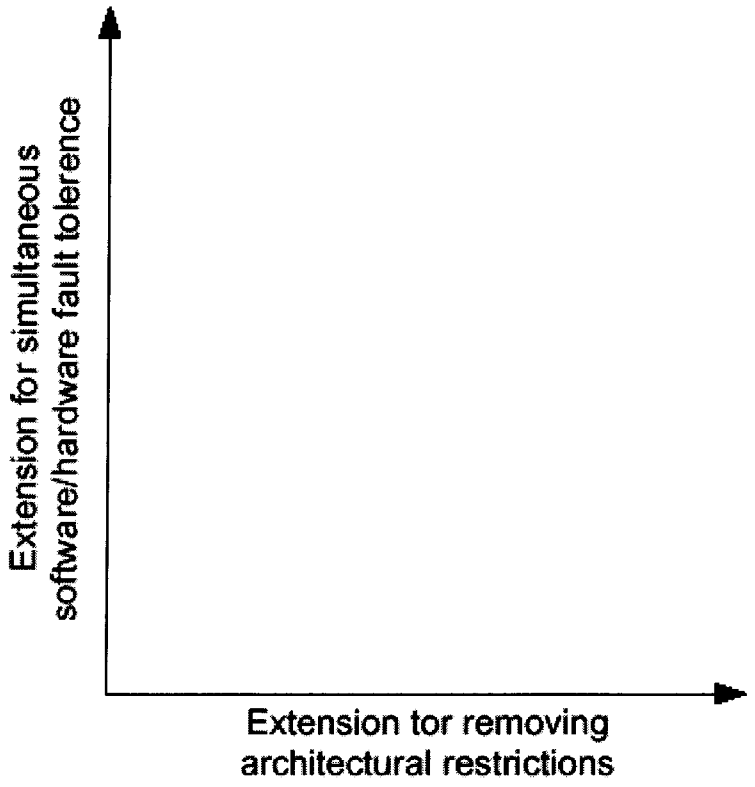


Sun/Solaris
Application Process 1: Attitude Control (new version)
GSU Middleware (P_1^{new})
Sun/Solaris
Application Process 1: Attitude Control (old version)
GSU Middleware (P_1^{old})
Sun/Solaris
Application Process 2: Attitude Determination
GSU Middleware (P_2)
Thruster
Star Tracker
Ethernet

Guarded Software Upgrade

“Dimensions” of Extension

We have extended the MDCD approach along two dimensions: 1) extension for simultaneous software/hardware fault tolerance, and 2) extension for removing architectural restrictions.


Extension for simultaneous software/hardware fault tolerence
Extension tor removing architectural restrictions

Presented for the Paper “Onboard Guarded Software Upgrading: Motivation, Approach, and Discussion”

Guarded Software Upgrade

Concluding Remarks

The message-driven confidence-driven nature of our approach makes it different from traditional software fault tolerance in the following respects:

- The approach does not impose restrictions on the interactions among application processes;
- The algorithms allow us to provide fault tolerance to a critical software component only, while letting other components be protected against the effect of error propagation;
- The dynamic error containment and recovery mechanisms are transparent to the application and thus can be implemented by generic middleware.

Direction of Dependable Computing Researches for Future Flight Missions

- Characteristics of Future Deep Space Missions:
  - Future Missions will Include
    - Multi-spacecraft formation flying (e.g., interferometry missions)
    - Planetary and Inter-planetary networks
    - Ultra reliable, ultra long life systems
    - Evolvable systems (e.g., landers and rovers) that can adapt to unexplored environments (e.g., surfaces of outer planet satellites)
  - New Missions requirements
    - High precision control for formation flying
    - Distributed processing among multiple spacecrafts
    - High speed wireless communication among spacecrafts
    - Mission time will be much longer than the life time of individual components
    - Much higher level of on-board autonomy

Direction of Dependable Computing Researches for Future Flight Missions

- Dependable Computing for Future Deep Space Missions
  - Need to tolerate the unreliable wireless communication links among spacecrafts
    - Employ mobile wireless network techniques
  - Need to tolerate spacecraft failures
    - Employ dependable computing techniques for large network
    - Problem of the threat of a tumbling or out-of-control spacecraft to other spacecrafts
  - Need to tolerate component failures due to aging
    - Improve the efficiency of redundancy techniques
    - On-board self-repair or salvaging techniques to reuse failed compoents
  - Need to incorporate evolvable computing techniques into the dependable computing system design.